MICROSOFT®

The High Performance Software™

Microsoft® FORTRAN Optimizing Compiler

Microsoft® CodeView™

MICROSOFT®

The High Performance Software™

Microsoft® CodeView™

Window-Oriented Debugger

for the MS-DOS® Operating System

Microsoft Corporation

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy this software on magnetic tape, disk, or any other medium for any purpose other than the purchaser's personal use.

© Copyright Microsoft Corporation, 1986, 1987

If you have comments about the software, complete the Software Problem Report at the back of this manual and return it to Microsoft Corporation.

If you have comments about the software documentation, complete the Documentation Feedback reply card at the back of this manual and return it to Microsoft Corporation.

Microsoft®, the Microsoft logo, and MS-DOS® are registered trademarks of Microsoft Corporation.
CodeView™ is a trademark of Microsoft Corporation.

AT&T® is a registered trademark of American Telephone and Telegraph Company.

Eagle® is a registered trademark of Eagle Computer Inc.

IBM® is a registered trademark of International Business Machines Corporation.

Intel® is a registered trademark of Intel Corporation.

Tandy® is a registered trademark of the Tandy Corporation.

Document No. 410500010-400-R01-0187
Part No. 005-014-040

Contents

1 Introduction to the Microsoft® CodeView™ Debugger 1

1.1 Introduction 3
1.2 Overview 3
1.3 About This Manual 4
1.4 Notational Conventions 6
1.5 Definitions 10
1.6 Learning Source-Level Features 11

2 Getting Started 13

2.1 Introduction 15
2.2 Starting the Sample Session 15
2.3 Preparing FORTRAN Programs for the CodeView Debugger 15
2.4 Starting the CodeView Debugger 19
2.5 Using CodeView Options 23
2.6 Using the CodeView Debugger with the Macro Assembler 33

3 The CodeView Display 35

3.1 Introduction 37
3.2 Using Window Mode 38
3.3 Using Sequential Mode 66

4 Using Dialog Commands 69

4.1 Introduction 71
4.2 Entering Commands and Arguments 71
4.3 Format for CodeView Commands and Arguments 73
4.4 Expressions 74

5 Executing Code 87

5.1 Introduction 89
5.2 Trace Command 90
5.3 Program Step Command 93
5.4 Go Command 95
5.5 Execute Command 98
5.6 Restart Command 99

6 Examining Data and Expressions 101

6.1 Introduction 103
6.2 Display Expression Command 103
6.3 Examine Symbols Command 109
6.4 Dump Commands 112
6.5 Register Command 122
6.6 8087 Command 123

7 Managing Breakpoints 127

7.1 Introduction 129
7.2 Breakpoint Set Command 129
7.3 Breakpoint Clear Command 132
7.4 Breakpoint Disable Command 134
7.5 Breakpoint Enable Command 135
7.6 Breakpoint List Command 136

8 Managing Watch Statements 139

8.1 Introduction 141
8.2 Setting Watch-Expression and Watch-Memory Statements 142
8.3 Setting Watchpoints 146
8.4 Setting Tracepoints 150
8.5 Deleting Watch Statements 155
8.6 Listing Watchpoints and Tracepoints 156

9 Examining Code 159

9.1 Introduction 161
9.2 Set Mode Command 161
9.3 Unassemble Command 163
9.4 View Command 166
9.5 Current Location Command 169
9.6 Stack Trace Command 170

10 Modifying Code or Data 173

10.1 Introduction 175
10.2 Assemble Command 175
10.3 Enter Commands 178
10.4 Register Command 190

11 Using System-Control Commands 193

11.1 Introduction 195
11.2 Help Command 195
11.3 Quit Command 196
11.4 Radix Command 197
11.5 Redraw Command 199
11.6 Screen Exchange Command 200
11.7 Search Command 201
11.8 Shell Escape Command 203
11.9 Tab Set Command 206
11.10 Redirection Commands 207

Appendixes 215

A Command and Mode Summary 217

A.1 Introduction 219
A.2 Modes 219
A.3 Options 221
A.4 Window Commands 222
A.5 Dialog Commands 224
A.6 Format Specifiers 227

B Regular Expressions 229

B.1 Introduction 231
B.2 Special Characters in Regular Expressions 231
B.3 Searching for Special Characters 232
B.4 Using the Period 232
B.5 Using Brackets 233
B.6 Using the Asterisk 234
B.7 Matching the Start or End of a Line 235

C Error Messages 237

Glossary 249

Index 259

Figures

Figure 1.1 CodeView Screen in Window Mode 9

Figure 2.1 CodeView Start-Up Screen in Window Mode 22

Figure 3.1 Elements of the CodeView Debugging Screen 38

Figure 3.2 The File Menu 51

Figure 3.3 The Search Menu 53

Figure 3.4 The View Menu 55

Figure 3.5 The Run Menu 57

Figure 3.6 The Watch Menu 58

Figure 3.7 The Options Menu 60

Figure 3.8 The Calls Menu 63

Figure 8.1 Watch-Command Statements in the Watch Window 146

Figure 8.2 Watchpoint-Command Statements in the Watch Window 149

Figure 8.3 Tracepoints in the Watch Window 154

Tables

Table 2.1	Default Exchange and Display Modes	28
Table 4.1	CodeView Operators	75
Table 4.2	FORTRAN Intrinsic Functions Supported by the CodeView Debugger	77
Table 4.3	CodeView Radix Examples	79
Table 4.4	Registers	80
Table 6.1	CodeView Format Specifiers	105
Table 10.1	Flag-Value Mnemonics	191
Table A.1	CodeView Modes	220
Table A.2	Window Commands	222
Table A.3	Menu Selections	223
Table A.4	Dialog Commands	225
Table A.5	Format Specifiers	228

Chapter 1

Introduction to the Microsoft® CodeView™ Debugger

1.1 Introduction 3
1.2 Overview 3
1.3 About This Manual 4
1.4 Notational Conventions 6
1.5 Definitions 10
1.6 Learning Source-Level Features 11

1.1 Introduction

The Microsoft® CodeView™ debugger is a debugging program that helps you test executable files developed with the Microsoft FORTRAN Optimizing Compiler. This chapter introduces you to the CodeView debugger, and summarizes the manual and the conventions used in it.

1.2 Overview

Important

You do *not* have to be an expert in assembly language to use the CodeView debugger; it brings the power of a full-featured debugging tool to high-level-language programmers. The debugger contains a full set of assembly-language-level features, but these are completely optional. When operating the CodeView debugger at the FORTRAN source level, you communicate with it using FORTRAN expressions, and the debugger displays the values of FORTRAN variables and expressions in decimal numbers.

If you wish to use only the debugger's source-level features, you will find some helpful suggestions in Section 1.6, "Learning Source-Level Features."

The CodeView debugger can display and execute program code, control program flow, and examine or change values in memory. Its window interface makes debugging easy. You can view your source code in one window, commands and responses in another, registers and flags in a third, and the values of variables or expressions in a fourth. You can examine the values of global or local variables, either by themselves or combined with other variables in expressions.

The window interface is designed for IBM® Personal Computers and IBM-compatible computers. However, you can also use the CodeView debugger with non-IBM-compatible computers using a sequential interface. Any debugging operation that can be performed with the window interface can also be performed with the sequential interface.

The CodeView debugger can access program locations through addresses, symbols, or line-number references. This makes it easy to locate and debug specific sections of code. You can debug programs at the source level, or you can examine code at the machine level in the debugger's assembly-language mode.

CodeView commands can be entered either from the keyboard or, in many cases, with the Microsoft Mouse (with the window interface only). Once you learn the commands, you can work most efficiently using both the mouse and the keyboard. The mouse is not required; all commands can be entered from the keyboard.

Note

The CodeView debugger is designed specifically for the Microsoft Mouse. Many manufacturers advertise their pointing devices as being compatible with the Microsoft Mouse. The CodeView debugger may work with some of these devices if they are closely compatible.

The CodeView debugger is simple to learn and use. Its commands are logical and easy to understand, especially for programmers who are familiar with the Microsoft Symbolic Debug Utility (**SYMDEB**) or the **DEBUG** utility provided with DOS. The CodeView user interface shares some features with its predecessors, but also incorporates powerful new features such as pop-up menus, multiple windows, mouse support, and single-keystroke commands.

1.3 About This Manual

This manual explains how to use the CodeView debugger to examine programs and locate program errors. It is a companion manual to the *Microsoft FORTRAN Compiler User's Guide* and the *Microsoft FORTRAN Compiler Language Reference*. See those manuals for information about developing FORTRAN programs.

Although the manual focuses on debugging FORTRAN programs using source code, the CodeView debugger can also debug assembly-language programs, or it can debug FORTRAN programs at the assembly-language level. If you are not familiar with assembly-language programming, some

aspects of the debugger may be unfamiliar (assembly mode and the register window, in particular).

As already noted, all CodeView features relating to assembly language are optional. You can simply ignore the assembly-mode features and debug your programs in the FORTRAN source mode. If you wish to learn more about assembly-language programming, you should consider purchasing the Microsoft Macro Assembler and reading its manual along with one of the many tutorial books on assembly-language programming.

The following list tells how to find information on various aspects of the CodeView debugger:

For This Information:	See:
Compiling and linking FORTRAN programs in the special format required by the CodeView debugger, and invoking the debugger with various command-line options	Chapter 2, "Getting Started"
Using elements of the CodeView display, including windows, pop-up menus, and the mouse	Chapter 3, "The CodeView Display"
Specifying arguments for dialog commands and using the CodeView operators to create expressions	Chapter 4, "Using Dialog Commands"
Executing all or part of your program	Chapter 5, "Executing Code"
Testing the value of expressions, or examining data of different sizes	Chapter 6, "Examining Data and Expressions"
Setting, enabling, disabling, clearing, and listing breakpoints	Chapter 7, "Managing Breakpoints"
Creating watch statements and managing the watch window	Chapter 8, "Managing Watch Statements"
Examining code and tracing function, procedure, or subroutine calls	Chapter 9, "Examining Code"
Modifying data or code in memory	Chapter 10, "Modifying Code or Data"
Controlling the operation of the CodeView debugger	Chapter 11, "Using System-Control Commands"

In addition to the information above, the following information is included in the appendixes:

For This Information:	See:
A summary of CodeView modes, commands, and menus	Appendix A, "Command and Mode Summary"
How to use regular expressions to find variable text strings in a source file	Appendix B, "Regular Expressions"
A list of CodeView error messages	Appendix C, "Error Messages"
Definitions of terms used in the manual	Glossary

Important

There may be additional information about the CodeView debugger in the **README.DOC** file. This file will describe any additions to the documentation or changes made to the program after the manual was printed.

1.4 Notational Conventions

The following notational conventions are used throughout this manual:

Example of Convention	Description of Convention
KEYWORDS and other concepts	Bold capital letters are used for the names of files, directories, registers, environment variables, FORTRAN keywords, and FORTRAN intrinsic functions. Commands typed at the DOS level are also capitalized. These commands include built-in DOS commands such as **SET**, as well as program names such as **FL** and **LINK**. You are not required to use capital letters when you actually enter DOS commands or FORTRAN keywords and intrinsic functions.

Bold type sometimes indicates text that must be typed exactly as shown. Text that must be typed as shown includes compiler options and FORTRAN operators.

Examples are shown below:

+	**INT4**
CONTINUE	**FUNCTION**
IF	**/Zi**

placeholders

Italics mark placeholders in command lines and option specifications. A placeholder represents a variable item that must appear at a specific point. Consider the command syntax for the Radix command:

N*number*

Note that *number* is italicized to indicate that it represents a general form for the Radix (**N**) command. In an actual command, the user supplies a particular number for the placeholder *number*.

Occasionally, italics may be used to emphasize particular words in the text.

Examples

Examples are displayed in a special nonproportional typeface so that they will look more like the programs you create with a text editor or the output of commonly used computer printers. If a command produces output, the input that you type is shown in boldface, while the output displayed by the CodeView debugger is shown in regular, nonboldface type, as in the following example:

```
>RAX
AX 0041
:43
>
```

```
Program
.
.
.
Fragment
```

Vertical ellipsis dots are used in program examples to indicate that a portion of the program has been omitted. For instance, in the following excerpt, three statements are shown. The ellipsis dots between the statements indicate that intervening program lines occur, but are not shown.

```
COUNT = 0
.
.
.
PASS = PASS + 1
.
.
.
COUNT = 0
```

[[*optional items*]]

Double brackets enclose optional fields in command-line and option syntax. Consider the following command-line syntax:

R [[*register*]] [[[=]]*value*]]

The double brackets around the placeholders indicate that you may enter a *register* and you may enter a *value*. The equal sign (=) in the second set of brackets indicates that you may place an equal sign before the *value*, but only if you specify a *value*.

[[*choice1* | *choice2*]]

The vertical bar used in conjunction with double brackets indicates that you may enter one of the entries shown on either side of the bar, as the following syntax block illustrates:

DB [[*address* | *range*]]

The bar indicates that following the Dump Bytes command (**DB**), you can specify either an *address* or a *range*. Since both are in double brackets, you can also give the command with no argument.

The vertical bar used in conjunction with curly braces indicates that you must enter one of the entries shown on either side of the bar, as the following syntax block illustrates:

W[[*type*]] {*address* | *range*}

"Quotation marks" — Quotation marks set off terms defined in the text. For example, the term "highlight" appears in quotation marks the first time it is defined.

KEY NAMES — Small capital letters are used for the names of keys and key sequences, such as ENTER, CONTROL-C, and ALT-F.

Sample screens — Sample screens are shown in black and white. Your screens will look like this if you have a monochrome monitor, or if you use the **/B** option in the CodeView command line (see Section 2.5.1 in Chapter 2, "Getting Started"). Figure 1.1 shows an example. Screens will be slightly different if you use a color monitor in color mode.

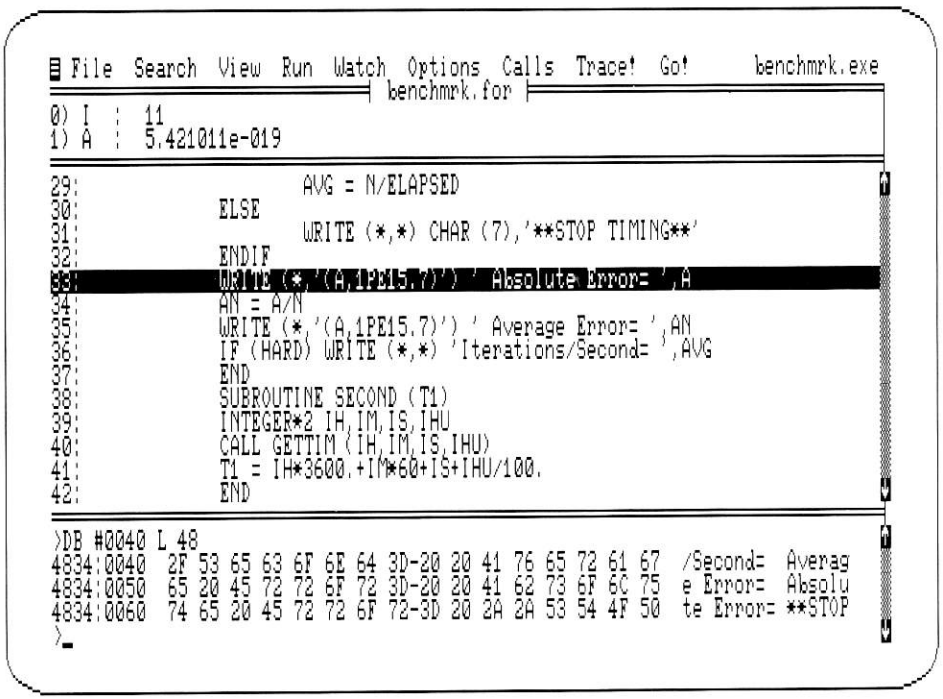
File Search View Run Watch Options Calls Trace! Go! benchmrk.exe
benchmrk.for
0) I : 11
1) A : 5.421011e-019
AVG = N/ELAPSED
ELSE
WRITE (*,*) CHAR (7),'**STOP TIMING**'
ENDIF
WRITE (*,'(A,1PE15.7)') ' Absolute Error= ',A
AN = A/N
WRITE (*,'(A,1PE15.7)') ' Average Error= ',AN
IF (HARD) WRITE (*,*) 'Iterations/Second= ',AVG
END
SUBROUTINE SECOND (T1)
INTEGER*2 IH,IM,IS,IHU
CALL GETTIM (IH,IM,IS,IHU)
T1 = IH*3600.+IM*60+IS+IHU/100.
END
>DB #0040 L 48
4834:0040 2F 53 65 63 6F 6E 64 3D-20 20 41 76 65 72 61 67 /Second= Averag
4834:0050 65 20 45 72 72 6F 72 3D-20 20 41 62 73 6F 6C 75 e Error= Absolu
4834:0060 74 65 20 45 72 72 6F 72-3D 20 2A 2A 53 54 4F 50 te Error= **STOP
>

Figure 1.1 CodeView Screen in Window Mode

1.5 Definitions

The CodeView debugger deals with programs on both the source and assembly-language levels. Several terms used in this manual mean slightly different things, depending on whether source or assembly-language features are being discussed. Wherever possible, this manual will use an equivalent generic term rather than one of these language-dependent terms.

Terms that have a special use in this manual or are language dependent are listed below:

Term	Meaning
DOS	A term used in Microsoft manuals to include both MS-DOS® and PC-DOS, except when noting features that are unique to one or the other.
Symbol	A name, often descriptive, that identifies a location anywhere in memory. In FORTRAN, the names of variables and subprograms (subroutines or functions) are symbols.
Label	In FORTRAN, a number that identifies a line in the source code (marking, for example, the end of a **DO** loop or the target of a **GOTO** statement). In assembly language, a label is a symbol that identifies an address in the code segment. The CodeView debugger does not recognize FORTRAN line labels. Unless otherwise stated, all references to labels in this manual refer to assembly-language labels.
Routine	Any block of code that is called from somewhere else in a program. After the routine is executed, program control returns to the statement following the calling statement. In FORTRAN, subprograms (subroutines or functions) are routines. Intrinsic functions are also routines (the call takes place on the assembly-language level to a routine in the FORTRAN libraries). Assembly-language routines are also called "procedures."

Call	A statement that temporarily transfers program control to a routine, as described above.
CodeView format specifier	A one-letter code that may be appended to an expression to control how the value of that expression will be formatted for display. These specifiers come from **printf**, a function in the standard library of the C language.

1.6 Learning Source-Level Features

The CodeView debugger has many features, all of which are described in detail in this manual. If you are unfamiliar with source-level debuggers, some of these features will be new to you. If you are interested in learning to use the CodeView debugger only at the source level, you can simply ignore those features that don't apply to your needs. Assuming that you can use the CodeView window mode, here are a few suggestions on how to proceed:

1. Run through the sample session provided on the distribution disk. This will give you a broad idea of the CodeView debugger's power and flexibility.
2. When you begin to experiment with the CodeView debugger on your own, use only the menu commands. Their use rapidly becomes self evident, especially if you use the mouse. The menu commands are described in Chapter 3, "The CodeView Display."
3. When first reading through this manual, don't worry about the dialog commands or memory-level and assembly-level features. Pay special attention to any text that describes how to use a CodeView command with expressions. Most of your work with the CodeView debugger at the source level will involve FORTRAN expressions and variable names.
4. Experiment with the sample-session program or another program you know well. Learn how to set and delete breakpoints, which stop program execution. Use the Go, Program Step, and Trace commands to execute parts of your program. Display some variables in the watch window. Watch the variables change as you execute different parts of your program. It is particularly instructive to set a breakpoint inside a loop and watch how the variables inside the loop change with each iteration. These tasks are explained in Chapters 4, 5, and 7.

5. Once you master breakpoints, learn how to use watchpoints and tracepoints, which stop program execution only when an expression becomes nonzero or changes. Watch statements are described in Chapter 8.

6. As you become more familiar with the CodeView debugger, try the Display Expression dialog command, which is entered on the CodeView command line with a question mark (?). The Display Expression command is described in Section 6.2.

7. Then graduate to the Dump commands. Learn how variables are stored in memory and how various Dump commands correspond to certain variable types. Experiment with the CodeView format specifiers. This knowledge will be valuable even at the source level, as it allows you to look simultaneously at all elements in an array. The Dump commands and format specifiers are explained in Chapter 6.

8. Learn the dialog commands that correspond to the menu commands you use frequently. They are often more powerful, and sometimes take less time to use than the menu commands. Dialog commands are described in Chapter 4.

Chapter 2
Getting Started

2.1 Introduction 15

2.2 Starting the Sample Session 15

2.3 Preparing FORTRAN Programs for the CodeView Debugger 15

2.3.1 Writing FORTRAN Source Code 16

2.3.2 Compiling Source Files for the CodeView Debugger 16

2.3.3 Linking Object Files for the CodeView Debugger 18

2.4 Starting the CodeView Debugger 19

2.5 Using CodeView Options 23

2.5.1 Starting with a Black-and-White Display 25

2.5.2 Specifying Start-Up Commands 26

2.5.3 Setting the Screen-Exchange Mode 27

2.5.4 Enabling Window or Sequential Mode 29

2.5.5 Turning Off the Mouse 30

2.5.6 Handling Interrupt Trapping 31

2.5.7 Using the Enhanced Graphics Adapter's 43-Line Mode 32

2.5.8 Using Two Video Adapters 32

2.6 Using the CodeView Debugger with the Macro Assembler 33

2.1 Introduction

Getting started with the CodeView debugger requires several simple steps. You must prepare a special-format executable file for the program you wish to debug; then you invoke the debugger. You may also wish to specify options that will affect the debugger's operation.

This chapter describes a disk-based tutorial session that shows you how to prepare a file for debugging and introduces you to the CodeView debugger. The chapter also describes how to compile and link a FORTRAN program to produce an executable file in the CodeView format, and how to load a program into the CodeView debugger. Using the debugger with assembly-language programs is also discussed briefly.

2.2 Starting the Sample Session

The sample program provided on the "Learning CodeView" disk illustrates many features of the CodeView debugger. You may wish to try this session before reading further in the manual.

To start the session, insert the "Learning CodeView" disk into a floppy disk drive, log on to the drive, and type `SAMPLE`.

2.3 Preparing FORTRAN Programs for the CodeView Debugger

Executable files must be in a special format in order to be used with the CodeView debugger. The special-format files contain line-number information and a symbol table in addition to executable code. You must use the correct options to put this additional information into the object files during compilation, and then into the executable file during linking.

If you try to debug an executable file that does not contain this additional information, the debugger will not be able to interpret symbols or to correlate code addresses and source line numbers. You can still debug your program in assembly mode, but many of the most powerful features of the CodeView debugger will be disabled.

The above restrictions also apply to programs compiled with versions of the Microsoft FORTRAN Compiler earlier than Version 4.0. To make full use of the CodeView debugger on older FORTRAN programs, you must recompile and relink them using the Microsoft FORTRAN Optimizing Compiler, Version 4.0 or later, and the Microsoft Overlay Linker, Version 3.5 or later. You can compile and link your programs using the **FL** command described in the *Microsoft FORTRAN Compiler User's Guide*. See the user's guide for complete instructions on compiling and linking. The special steps required for compiling and linking programs for the CodeView debugger are explained in Sections 2.3.1 – 2.3.3 of this manual.

2.3.1 Writing FORTRAN Source Code

Any source code that is legal in FORTRAN can be compiled to an executable file and debugged with the CodeView debugger. However, some programming practices make debugging more difficult. You should be aware of how these practices affect debugging.

For example, the FORTRAN language permits you to put code in separate include files and then read the files into your source file using the **$INCLUDE** metacommand. However, you will not be able to use the CodeView debugger to debug source code in include files. The preferred method is to write and compile separate library modules, then link the resulting object files with your programs. The CodeView debugger does support this technique, and will automatically display the correct source file when you trace into a routine contained in a separately compiled file.

2.3.2 Compiling Source Files for the CodeView Debugger

Note

The **FL** control program will accept command-line options that are preceded by either a forward slash (**/**) or a dash (**-**). For brevity, this manual will list only the forward slash when describing options, but you may use either symbol. Also, though file names may be uppercase or lowercase, the letter case of **FL** command-line options is significant. You must type the letters exactly as given.

When you compile a source file for a program you want to debug, you must specify the **/Zi** option on the **FL** command line. The **/Zi** option instructs the compiler to include line-number and symbolic information in the object file.

If you do not need complete symbolic information in some modules, you can compile those modules with the **/Zd** option instead of **/Zi**. The **/Zd** option writes less symbolic information to the object file, so using this option will save disk space and memory. For example, if you are working on a program made up of five modules, but you only need to debug one module, you can compile that module with the **/Zi** option and the other modules with the **/Zd** option, then link them to create an executable file. (See Chapter 4 of the *Microsoft FORTRAN Compiler User's Guide* for information about linking separately compiled modules.) The executable file will be significantly smaller than if you had compiled all modules with **/Zi**. You will be able to examine global variables and see source lines in modules compiled with the **/Zd** option, but local variables will be unavailable.

In addition, you will probably want to specify the **/Od** option. This option turns off optimization. Optimized code may be rearranged for greater efficiency and, as a result, the instructions in your program may not correspond closely to the source lines. Variables, especially in loops, may be temporarily assigned to registers, or not be referenced at all, so Watch and Trace commands may not show the appropriate value. After debugging, you can compile a final version of the program with the optimization level you prefer.

You cannot debug a program until you compile it successfully. The CodeView debugger will not help you correct syntax or compiler errors. You should refer to a source listing and the appropriate reference books to correct your code until you can compile it successfully; then use the debugger to locate logical errors in the program. Finally, you can correct these errors in the source code and recompile.

Note

The CodeView debugger cannot be used on modules processed with the Microsoft Library Manager (**LIB**), even when they are compiled with the **/Zi** option. **LIB** assumes the modules it processes have already been debugged, and strips all debugging information.

■ **Example**

```
FL /Zi /Od EXAMPLE.FOR
```

The above example compiles the source file **EXAMPLE.FOR** to produce an object file called **EXAMPLE.OBJ**. The object file contains line-number information, a symbol table, and unoptimized object code. **FL** then automatically invokes the linker, which forms an executable file, **COUNT.EXE**, containing the symbolic and line-number information required by the CodeView debugger.

2.3.3 Linking Object Files for the CodeView Debugger

If you use **LINK** separately (instead of invoking it through **FL**) to link an object file or files for debugging, you should specify the **/CODEVIEW** option (it can be abbreviated as **/CO**). This instructs the linker to incorporate addresses for symbols and source lines into the executable file. Note that when you use the **FL** command, you do not need to specify the **/CODEVIEW** link option, since the program automatically supplies this option when you specify the **/Zi** compile option.

No other option is necessary for the debugger, but you may use other options if your program requires them (see the *Microsoft FORTRAN Compiler User's Guide*). For example, you could use the **/MAP** or **/PAUSE** option.

Although executable files prepared with the **/CODEVIEW** option can be executed from the DOS command line like any other executable files, they are larger because of the extra symbolic information in them. To minimize program size, you will probably want to recompile and link your final version without the **/Zi** option when you finish debugging a program.

■ **Examples**

```
LINK /CO EXAMPLE
```

The above example links the object file **EXAMPLE.OBJ** to form an executable file, **EXAMPLE.EXE**, containing the symbolic and line-number information required by the CodeView debugger.

```
FL /Zi EXAMPLE
```

The example shown above uses the **FL** command to link the object file **EXAMPLE.OBJ** to form an executable file, **EXAMPLE.EXE**. (Note that the extension has been omitted from the name of the file to be linked, so compilation is skipped.)

2.4 Starting the CodeView Debugger

Before starting the debugger, make sure all the files it requires are available in the proper places. The following files are recommended for FORTRAN source-level debugging:

File	Location
CV.EXE	The CodeView program file can be in the current directory, or in any directory accessible with the **PATH** command. For example, if you set up your FORTRAN compiler files on a hard disk using the **SETUP** program provided on the distribution disk, you might put **CV.EXE** in the **\BIN** directory.
CV.HLP	If you want to have the on-line help available during your session, you should have this file either in the current directory, or in any directory accessible with the **PATH** command. For example, if you set up your FORTRAN compiler files on a hard disk using the **SETUP** program provided on the distribution disk, you might put **CV.HLP** in the **\BIN** directory. If the CodeView debugger cannot find the help file, you can still use the debugger, but you will see an error message if you try to use one of the help commands.
program.**EXE**	The executable file for the program you wish to debug must be in the current directory, or in a drive and directory you specify as part of the start-up file specification. The CodeView debugger will display an error message and refuse to start if the executable file is not found.

program.**FOR**

Normally, source files should be in the current directory. However, if you specify a file specification for the source file during compilation, that specification will become part of the symbolic information stored in the executable file. For example, if you compiled with the command line `FL -Zi -Od DEMO.FOR`, the CodeView debugger will expect the source file to be in the current directory. However, if you compiled with the command line `FL -Zi -Od \SOURCE\DEMO.FOR`, then the debugger will expect the source file to be in directory `\SOURCE`. If the debugger cannot find the source file in the directory specified in the executable file (usually the current directory), the program will prompt you for a new directory. You can either enter a new directory, or you can press the ENTER key to indicate that you do not want a source file to be used for this module. If no source file is specified, you must debug in assembly mode.

If the appropriate files are in the correct directories, you can enter the CodeView command line at the DOS command prompt. The command line has the following form:

CV [[*options*]] *executablefile* [[*arguments*]]

The *options* are one or more of the options described in Section 2.5. The *executablefile* is the name of an executable file to be loaded by the debugger. It must have the extension **.EXE** or **.COM**. If you try to load a nonexecutable file, the following message appears:

```
Not an executable file
```

FORTRAN programs and assembly-language programs containing CodeView symbolic information will always have the extension **.EXE**. Files with the extension **.COM** can be debugged in assembly mode, but they can never contain symbolic information. Programs that use overlays cannot be debugged with the CodeView debugger.

The optional *arguments* are parameters passed to the *executablefile*. If the program you are debugging does not accept command-line arguments, you do not need to pass any arguments.

If you specify the *executablefile* as a file name with no extension, the CodeView debugger searches for a file with the given base name and the extension **.EXE**. Therefore, you must specify the **.COM** extension if you are debugging a **.COM** file. If the file you specify is not in the CodeView format, the debugger starts in assembly mode and displays the following message:

```
No symbolic information
```

You must specify an executable file when you start the CodeView debugger. If you omit the executable file, the debugger displays a message showing the correct command-line format.

When you give the debugger a valid command line, the executable program and the source file are loaded, the address data is processed, and the CodeView display appears. The initial display will be in window mode or sequential mode, depending on the options you specify and the type of computer you have.

For example, if you wanted to debug the program **BENCHMRK.EXE**, you could start the debugger with the following command line:

```
CV BENCHMRK
```

If you give this command line on an IBM Personal Computer, window mode will be selected automatically. The display will look like Figure 2.1.

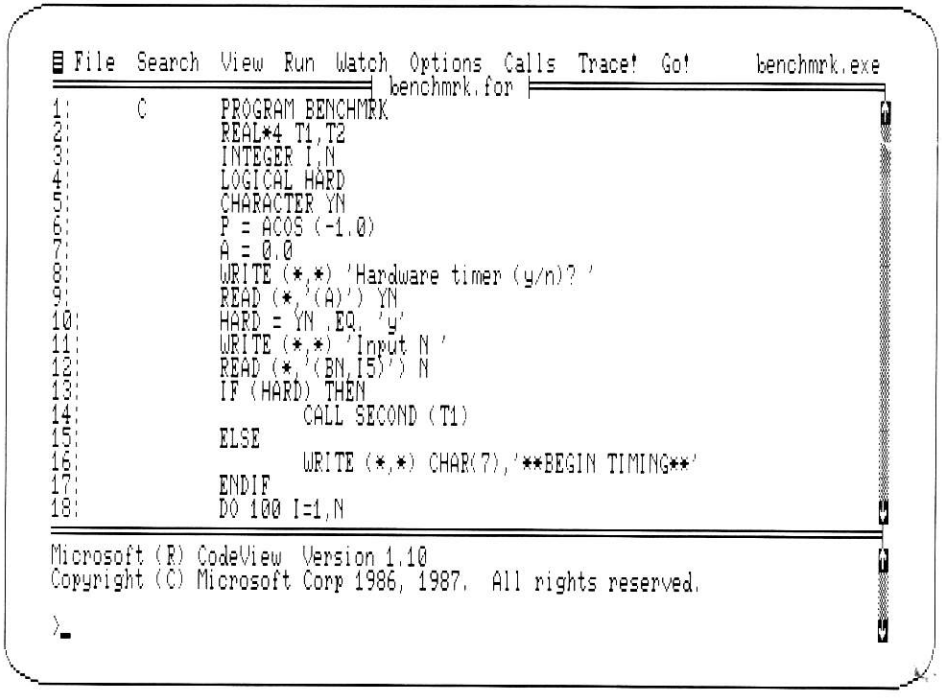
File Search View Run Watch Options Calls Trace! Go! benchmrk.exe
benchmrk.for
1: C PROGRAM BENCHMRK
2: REAL*4 T1,T2
3: INTEGER I,N
4: LOGICAL HARD
5: CHARACTER YN
6: P = ACOS (-1.0)
7: A = 0.0
8: WRITE (*,*) 'Hardware timer (y/n)? '
9: READ (*,'(A)') YN
10: HARD = YN .EQ. 'y'
11: WRITE (*,*) 'Input N '
12: READ (*,'(BN,I5)') N
13: IF (HARD) THEN
14: CALL SECOND (T1)
15: ELSE
16: WRITE (*,*) CHAR(7),'**BEGIN TIMING**'
17: ENDIF
18: DO 100 I=1,N
Microsoft (R) CodeView Version 1.10
Copyright (C) Microsoft Corp 1986, 1987. All rights reserved.
>

Figure 2.1 CodeView Start-Up Screen in Window Mode

If you give the same command line on most non-IBM computers, sequential mode will be selected. The following lines appear:

```
Microsoft (R) CodeView  Version 1.10
Copyright (C) Microsoft Corp 1986, 1987. All rights reserved.

>
```

You can use CodeView options, as described in Section 2.5, to override the default start-up mode.

If your program is written in FORTRAN, the CodeView debugger is now at the beginning of the start-up code that precedes your program. In source mode, you can enter an execution command (such as Trace or Program Step) to automatically execute through the start-up code to the beginning of your program. At this point, you are ready to start debugging your program, as described in Chapters 3 – 11.

2.5 Using CodeView Options

You can change the start-up behavior of the debugger by specifying options in the command line.

An option is a sequence of characters preceded by either a forward slash (/) or a dash (-). For brevity, this manual will list only the forward slash when describing options, but you may use either. Unlike the **FL** command-line options, CodeView command-line options are not case sensitive.

A file containing a dash in its file name must be renamed before use with the CodeView debugger so that the debugger will not interpret the dash as an option designator. You can use more than one option in a command line, but each option must have its own option designator and spaces must separate each option from other elements of the command line.

Note

The CodeView debugger's defaults are different for IBM Personal Computers than for other computers. However, the debugger may not always recognize the difference between computers. The debugger determines if you have an IBM computer by looking at certain locations in memory.

The following list suggests some situations in which you might want to use an option. If more than one condition applies, you can use more than one option (in any order). If none of the conditions applies, you need not use any options.

If:	Type:
You have an IBM-compatible computer and you want to use window mode	**/W**
You have a two-color monitor, a color graphics adapter (CGA), and an IBM or IBM-compatible computer	**/B**
You are debugging a graphics program and you want to be able to see the output screen	**/S**

You are debugging a program that uses multiple video-display pages and you want to be able to see the output screen	**/S**
You are using a non-IBM-compatible computer and you want to be able to see the output screen	**/S**
You are using an IBM-compatible computer to debug a program that does not use graphics or multiple video-display pages and you want to be able to see the output screen	**/F**
You are using an IBM-compatible computer that does not support certain IBM-specific interrupt trapping functions	**/D**
You are using a non-IBM-compatible computer and you want to enable CONTROL-C and CONTROL-BREAK	**/I**
You have an IBM computer, but you wish to debug in sequential mode (for example, with redirection)	**/T**
You have a mouse installed in your system, but you do not want to use it during the debugging session	**/M**
You want a 43-line display and you have an IBM or IBM-compatible computer with an enhanced graphics adapter (EGA) and an enhanced color display	**/43**
You want to use two monitors with the CodeView debugger	**/2**
You want the CodeView debugger to automatically execute a series of commands when it starts up	**/C***commands*

For example, assume you are using an IBM-compatible computer with a CGA and a two-color monitor. The program you are debugging, which you could name GRAPHIX.EXE, plots points in graphics mode. You want to be able to see the output screen during the debugging session. Finally, you want to be able to start the debugger several times without having to remember all the options, and you want to execute the FORTRAN start-up code automatically each time. You could create a batch file called STATSBUG.BAT consisting of the following line:

```
CV /W /B /S /CGmain GRAPHIX
```

The CodeView options are described in more detail in Sections 2.5.1 – 2.5.8.

2.5.1 Starting with a Black-and-White Display

■ Option

/B

The **/B** option forces the CodeView debugger to display in two colors even if you have a CGA. By default, the debugger checks on start-up to see what kind of display adapter is attached to your computer. If the debugger detects a monochrome adapter (MA), it displays in two colors. If it detects a CGA, it displays in multiple colors.

If you use a two-color monitor with a CGA, you may want to disable color. Monitors that display in only two colors (usually green and black, or amber and black) often attempt to show colors with different cross-hatching patterns, or in gray-scale shades of the display color. In either case, you may find the display easier to read if you use the **/B** option to force black-and-white display. Most two-color monitors still have four color distinctions: background (black), normal text, high-intensity text, and reverse-video text.

■ Example

```
CV /B CALC CALC.DAT
```

The above example starts the CodeView debugger in black-and-white mode. This is the only mode available if you have a monochrome adapter. The display is usually easier to read in this mode if you have a CGA and a two-color monitor.

2.5.2 Specifying Start-Up Commands

■ Option

/Ccommands

The **/C** option allows you to specify one or more *commands* that will be executed automatically upon start-up. You can use these options to invoke the debugger from a batch or **MAKE** file. Each command is separated from the previous command by a semicolon.

If one or more of your start-up commands has arguments that require spaces between them, you should enclose the entire option in double quotation marks. Otherwise, the debugger will interpret each argument as a separate CodeView command-line argument rather than as a debugging-command argument.

Warning

Any start-up option that uses the less-than (<) or greater-than (>) symbol must be enclosed in double quotation marks even if it does not require spaces. This ensures that the redirection command will be interpreted by the CodeView debugger rather than by DOS.

■ Examples

```
CV /CGmain CALC CALC.DAT
```

The above example loads the CodeView debugger with `CALC` as the executable file and `CALC.DAT` as the argument. Upon start-up, the debugger executes the FORTRAN start-up code with the command `Gmain`. Since no space is required between the CodeView command (**G**) and its argument (**main**), the option is not enclosed in double quotation marks.

```
CV "/C;S&;G INTEGRAL;DS ARRAYX L 20" CALC CALC.DAT
```

The example shown above loads the same file with the same argument as the first example, but the command list is more extensive. The debugger starts in mixed source/assembly mode (`S&`). It executes to the routine

INTEGRAL (G INTEGRAL), then dumps 20 short real numbers (REAL*4), starting at the address of the variable ARRAYX (DS ARRAYX L 20). Since several of the commands use spaces, the entire option is enclosed in double quotation marks.

```
CV "/C<INPUT.FIL" CALC CALC.DAT
```

The above example loads the same file and argument as the first example, but the start-up command directs the debugger to accept input from the file INPUT.FIL rather than from the keyboard. Although the option does not include any spaces, it must be enclosed in double quotation marks so that the less-than symbol will be read by the CodeView debugger rather than by DOS.

2.5.3 Setting the Screen-Exchange Mode

- **Options**

/F
/S

The CodeView debugger allows you to move quickly back and forth between the output screen, which contains the output from your program, and the debugging screen, which contains the debugging display. The debugger can handle this screen exchange in two ways: screen flipping or screen swapping. The **/F** option (screen flipping) and the **/S** option (screen swapping) allow you to choose the method from the command line.

If neither method is specified (possible only on non-IBM computers), the Screen Exchange command will not work. No screen exchange is the default for non-IBM computers. Screen flipping is the default for IBM computers with graphics adapters, and screen swapping is the default for IBM computers with monochrome adapters.

Screen flipping uses the video-display pages of the graphics adapter to store each screen of text. Video-display pages are a special memory buffer reserved for multiple screens of video output. This method is faster and uses less memory than screen swapping. However, screen flipping cannot be used with an MA, or to debug programs that produce graphics or use the video-display pages. In addition, the CodeView debugger's screen flipping works only with IBM and IBM-compatible microcomputers.

Screen swapping has none of the limitations of screen flipping, but is significantly slower and requires more memory. In the screen-swapping method, the CodeView debugger creates a buffer in memory and uses it to store the screen that is not being used. When the user requests the other screen, the debugger swaps the screen in the display buffer for the one in the storage buffer.

When you use screen swapping, the buffer size is 16K for all adapters. The amount of memory used by the CodeView debugger is increased by the size of the buffer.

Table 2.1 shows the default exchange mode (swapping or flipping) and the default display mode (sequential or window) for various configurations. Display modes are discussed in Section 2.5.4, "Enabling Window or Sequential Mode."

Table 2.1

Default Exchange and Display Modes

Computer	Display Adapter	Default Modes	Alternate Modes
IBM	CGA or EGA	**/F /W**	**/S** if your program uses video-display pages or graphics; **/T** for sequential mode
IBM compatible	CGA or EGA	**/T**	**/W** for window mode; **/F** for screen flipping with text programs, or **/S** for screen swapping with programs that use video-display pages or graphics
IBM	MA	**/S /W**	**/T** for sequential mode
IBM compatible	MA	**/T**	**/W** for window mode; **/S** for screen swapping
Noncompatible	Any	**/T**	**/S** for screen swapping

If you are not sure if your computer is completely IBM compatible, you can experiment. If the basic input/output system (BIOS) of your computer is not compatible enough, the CodeView debugger may not work with the **/F** option.

If you specify the **/F** option with an MA, the debugger will ignore the option and use screen swapping. If you try to use screen flipping to debug a program that produces graphics or uses the video-display pages, you may get unexpected results and have to start over with the **/S** option.

■ Examples

```
CV /F CALC CALC.DAT
```

The above example starts the CodeView debugger with screen flipping. You might use this command line if you have an IBM-compatible computer, and you want to override the default screen-exchange mode in order to use less memory and switch screens more quickly. The option would not be necessary on an IBM computer, since screen flipping is the default.

```
CV /S GRAFIX
```

The above example starts the debugger with screen swapping. You might use this command line if your program uses graphics mode.

2.5.4 Enabling Window or Sequential Mode

■ Options

/T
/W

The CodeView debugger can operate in window mode or in sequential mode. Window mode displays up to four windows, enabling you to see different aspects of the debugging-session program simultaneously. You can also use a mouse in window mode. Window mode requires an IBM or IBM-compatible microcomputer.

Sequential mode works with any computer, and is useful with redirection commands. Debugging information is displayed sequentially on the screen.

The behavior of each mode is discussed in detail in Chapter 3, "The CodeView Display." Refer back to Table 2.1 for the default and alternate modes for your computer. If you are not sure if your computer is completely IBM compatible, you can experiment with the options. If the BIOS of your computer is not compatible enough, you may not be able to use window mode (the **/W** option).

Note

Although window mode is more convenient, any debugging operation that can be done in window mode can also be done in sequential mode.

■ **Examples**

```
CV /W SIEVE
```

The above example starts the CodeView debugger in window mode. You will probably want to use the **/W** option if you have an IBM-compatible computer, since the default sequential mode is less convenient for most debugging tasks.

```
CV /T SIEVE
```

The above example starts the debugger in sequential mode. You might want to use this option if you have an IBM computer and you have a specific reason for using sequential mode. For instance, sequential mode usually works better if you are redirecting your debugging output to a remote terminal.

2.5.5 Turning Off the Mouse

■ **Option**

/M

If you have a mouse installed on your system, you can tell the CodeView debugger to ignore it, using the **/M** option. You may need to use this option if you are debugging a program that uses the mouse and your mouse is not a Microsoft Mouse. This is due to a conflict between the program's use of the mouse and the debugger's use of it. If you use the **/M** option, the program you are debugging can still use the mouse, but the CodeView debugger cannot.

Important

The same conflict between program and debugger applies if you are not using the version of the Microsoft Mouse driver programs (**MOUSE.SYS** and **MOUSE.COM**) included on the Microsoft FORTRAN Optimizing Compiler distribution disks. You may want to replace your old mouse driver program with the updated version. You will then be able to use the mouse with both the CodeView debugger and the program you are debugging. If you did not install a mouse driver when you set up Microsoft FORTRAN Optimizing Compiler Version 4.0, see Chapter 2 of your *Microsoft FORTRAN Compiler User's Guide* for information on installing **MOUSE.SYS** and **MOUSE.COM**. These programs will not work with pointing devices from other manufacturers.

2.5.6 Handling Interrupt Trapping

■ Options

/D
/I

The **/D** option turns off nonmaskable interrupt (NMI) and 8259 interrupt trapping. If you are using an IBM PC Convertible, a Tandy® 1000, or the AT&T® 6300 Plus and you are experiencing system crashes while using the CodeView debugger, try starting with the **/D** option. To enable window mode, use **/W** with **/D**; otherwise sequential mode is set automatically. Note that because this option turns off interrupt trapping, CONTROL-C and CONTROL-BREAK will not work, and an external interrupt may occur during a trace operation. If this happens you may find yourself tracing the interrupt handler instead of your program.

The **/I** option forces the debugger to handle NMI and 8259 interrupt trapping. Use this option to enable CONTROL-C and CONTROL-BREAK on computers the debugger does not recognize as being IBM compatible, such as the Eagle® PC. Window mode is set automatically with the **/I** option; you don't have to specify **/W**. Using the **/I** option lets you stop program execution at any point while you are using the CodeView debugger.

2.5.7 Using the Enhanced Graphics Adapter's 43-Line Mode

■ Option

/43

If you have an EGA and a monochrome monitor or an enhanced color display monitor (or a compatible monitor), you can use the **/43** option to enable a 43-line-by-80-column text mode. You cannot not use this mode with other monitors, or if you have a CGA or an MA. The CodeView debugger will ignore the option if it does not detect an EGA.

The EGA's 43-line mode performs identically to the normal 80-column-by-25-line mode used by default on the EGA, CGA, and MA. The advantage of the 43-line mode is that more text fits on the CodeView display; the disadvantage is that the text is smaller and harder to read. If you have an EGA, you can experiment to see which size you prefer.

■ Example

```
CV /43 CALC CALC.DAT
```

The above example starts the CodeView debugger in 43-line mode if you have an EGA video adapter and an enhanced color or monochrome monitor. The option will be ignored if you do not have the hardware to support it.

2.5.8 Using Two Video Adapters

■ Option

/2

The **/2** option permits the use of two monitors with the CodeView debugger. The program display will appear on the current default monitor, while the CodeView display appears on the other monitor. You must have two monitors and two adapters to use the **/2** option. For example, if you have both a color graphics adapter and a monochrome adapter, you might want to set the CGA up as the default adapter. You could then debug a graphics program with the graphics display appearing on the graphics monitor and

the debugging display appearing on the monochrome monitor. Microsoft Mouse support will be disabled on the debugging display if you use this option.

2.6 Using the CodeView Debugger with the Macro Assembler

You can use the CodeView debugger with files developed with the Microsoft (or IBM) Macro Assembler. Since the Microsoft Macro Assembler (Versions 1.0 through 4.0) does not write line numbers to object files, some of the CodeView debugger's features will not be used when you debug programs developed with the assembler.

The debugger can be used on either **.EXE** or **.COM** files, but you can only view symbolic information in **.EXE** files. The procedure for assembling and debugging **.EXE** files is summarized below:

1. In your source file, declare public any symbols, such as labels and variables, that you want to reference in the debugger. If the file is small, you may want to declare all symbols public.
2. Assemble as usual. No special options are required, and all assembly options are allowed.
3. Link with the linker provided with Version 4.0 of the Microsoft FORTRAN Optimizing Compiler. Do not use the linker provided with versions 4.0 and earlier of the Macro Assembler. Use the **/CODEVIEW** option when linking.
4. Debug in assembly mode (this is the start-up default if the debugger doesn't find line-number information). You cannot use source mode for debugging, but you can load the source file into the display window and view it in source mode. You may find this convenient for referring to macros and comments. Any labels or variables that you declared public in the source file can be displayed and referenced by name instead of by address.

You can also use this procedure to debug assembly-language modules called by your program.

The CodeView debugger is more limited when debugging assembly-language programs than when debugging FORTRAN programs. The following limitations apply:

- When writing stand-alone assembly-language programs that will be debugged with the CodeView debugger, the class type for the code segment must be CODE. For example, declare segments as follows:

```
sseg            SEGMENT para stack 'STACK'
sseg            ENDS

dseg            SEGMENT word public 'DATA'
dseg            ENDS

cseg            SEGMENT byte public 'CODE'
cseg            ENDS
```

- In programs assembled with Macro Assembler Versions 1.0 through 4.0, you cannot use variables in CodeView expressions because information on the type size of each variable is not written to the object file. You can still use constants. This means that the Display Expression command, the Watchpoint command, and the expression version of the Watch and Tracepoint commands do not work for such programs. Use the Dump command, and the memory versions of the Watch and Tracepoint commands, instead.

  For example, assume your program has the following declaration:

```
var_a   DW   30d
```

  You can use the following commands with this variable:

```
DI var_a L 1  ;* Dump var_a as an integer
WI var_a L 1  ;* Watch var_a as an integer
TPI var_a     ;* Break when var_a changes (1 is default)
```

- Assembly-language symbol names are normally converted to uppercase when assembled. If symbols in your source file are lowercase, you may want to turn off Case Sense from the Options menu.

Chapter 3
The CodeView Display

3.1 Introduction 37

3.2 Using Window Mode 38

3.2.1 Executing Window Commands with the Keyboard 40

3.2.1.1 Moving the Cursor with Keyboard Commands 40

3.2.1.2 Changing the Screen with Keyboard Commands 42

3.2.1.3 Controlling Program Execution with Keyboard Commands 43

3.2.1.4 Selecting from Menus with the Keyboard 44

3.2.2 Executing Window Commands with the Mouse 46

3.2.2.1 Changing the Screen with the Mouse 46

3.2.2.2 Controlling Program Execution with the Mouse 47

3.2.2.3 Selecting from Menus with the Mouse 49

3.2.3 Using Menu Selections 51

3.2.3.1 Using the File Menu 51

3.2.3.2 Using the Search Menu 53

3.2.3.3 Using the View Menu 55

3.2.3.4 Using the Run Menu 56

3.2.3.5 Using the Watch Menu 58

3.2.3.6 Using the Options Menu 60

3.2.3.7 Using the Calls Menu 63

3.2.4 Using the Help System 65

3.3 Using Sequential Mode 66

3.1 Introduction

The CodeView screen display can appear in two different modes: window and sequential. Either mode provides a useful debugging environment, but the window mode is the more powerful and convenient of the two.

Most users will prefer to use window mode, if they have the hardware to support it. In window mode, pop-up menus, function keys, and mouse support offer fast access to the most common commands. Different aspects of the program and debugging environment can be seen in different windows simultaneously. Window mode is described in Section 3.2.

Sequential mode should be familiar to most programmers. It is similar to the display mode of the CodeView debugger's predecessors, the Microsoft Symbolic Debug Utility (**SYMDEB**) and the DOS **DEBUG** utility. This mode is required if you do not have an IBM-compatible computer, and it is sometimes useful when redirecting command input or output. Sequential mode is described in Section 3.3.

3.2 Using Window Mode

Figure 3.1 shows the CodeView window-mode display with all windows open.

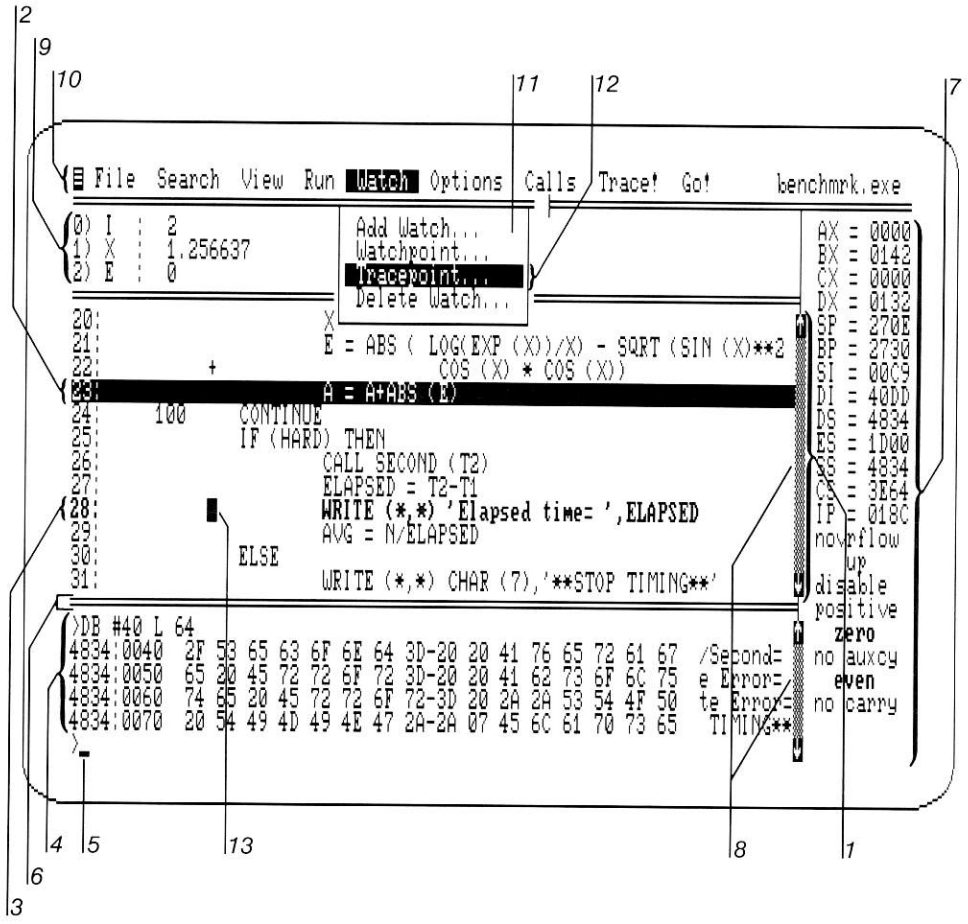

Figure 3.1 Elements of the CodeView Debugging Screen

The elements of the CodeView display marked in Figure 3.1 are explained below:

1. The display window shows the program being debugged. It can contain source code (as in the example), assembly-language instructions, or any specified text file.
2. The current location line (the next line the program will execute) is displayed in reverse video or in a different color. This line may not always be visible, since you can scroll to earlier or later parts of the program.
3. Lines containing previously set breakpoints are shown in high-intensity text.
4. The dialog window is where you enter dialog commands. These are the commands with optional arguments that you can enter at the CodeView prompt (>). You can scroll up or down in this window to view previous dialog commands and command output.
5. The cursor is a thin, blinking line that shows the location at which you can enter commands from the keyboard. You can move the cursor up and down, and put it in either the dialog or display window.
6. The display/dialog separator line divides the dialog window from the display window. You can move this line up or down to change the relative size of the two windows.
7. The register window shows the current status of the registers and flags. This is an optional window that can be opened or closed with one keystroke.
8. The scroll bars are the vertical bars on the right side of the screen. Each scroll bar has an up arrow and a down arrow that you can use to scroll through the display with a mouse.
9. The watch window is an optional window that shows the current status of specified variables or expressions. It appears automatically whenever you create watch statements. See Chapter 8, "Managing Watch Statements."
10. The menu bar shows titles of menus and commands that you can activate with the keyboard or the mouse. Titles followed by an exclamation point represent commands; other titles are menus.
11. Menus can be opened by specifying the appropriate title on the menu bar. On the sample screen, the Watch menu has been opened.
12. The menu "highlight" is a reverse-video or colored strip indicating the current selection in a menu. You can move the highlight up or down to change the current selection.

13. The mouse pointer indicates the current position of the mouse. It is shown only if you have a mouse installed on your system.
14. Dialog boxes (not shown) appear in the center of the screen when you choose a menu selection that requires a response. The box prompts you for a response and disappears when you enter your answer.
15. Message boxes (not shown) appear in the center of the screen to display errors or other messages.

The screen elements are described in more detail in the rest of this chapter.

3.2.1 Executing Window Commands with the Keyboard

The CodeView debugger accepts two kinds of commands: window commands and dialog commands. Dialog commands are entered as command lines following the CodeView prompt (>) in sequential mode. They are discussed in Chapter 4, "Using Dialog Commands."

The most common CodeView debugging commands and all the commands for managing the CodeView display are available with window commands. Window commands are one-keystroke commands that can be entered with function keys, CONTROL-key combinations, ALT-key combinations, or the direction keys on the numeric keypad.

Most window commands can also be entered with a mouse, as described in Section 3.2.2. The window commands available from the keyboard are described by category in Sections 3.2.1.1 – 3.2.1.4. For a table of commands by name, see Appendix A, "Command and Mode Summary."

3.2.1.1 Moving the Cursor with Keyboard Commands

The following keys move the cursor or scroll text up or down in the display or dialog window:

Key	Function
F6	Moves the cursor between the display and dialog windows. If the cursor is in the dialog window when you press F6, it will move to its previous position in the display window. If the cursor is in the display window, it will move to its previous position in the dialog window.

CONTROL-U	Moves the display/dialog separator line up one line. This decreases the size of the display window and increases the size of the dialog window. If the cursor is in the dialog window, you can remove the display window entirely by moving the separator line to the top of the window.
CONTROL-D	Moves the display/dialog separator line down one line. This increases the size of the display window and decreases the size of the dialog window. If the cursor is in the display window, you can remove the dialog window entirely by moving the separator line to the bottom of the screen.
UP ARROW	Moves the cursor up one line in either the display or dialog window.
DOWN ARROW	Moves the cursor down one line in either the display or dialog window.
PGUP	Scrolls up one page. If the cursor is in the display window, the source lines or assembly-language instructions scroll up. If the cursor is in the dialog window, the buffer of commands entered during the session scrolls up. The cursor remains at its current position in the window. The length of a page is the current number of lines in the window.
PGDN	Scrolls down one page. If the cursor is in the display window, the source lines or assembly-language instructions scroll down. If the cursor is in the dialog window, the buffer of commands entered during the session scrolls down. The cursor remains at its current position in the window. The length of a page is the current number of lines in the window.
HOME	Scrolls to the top of the file or command buffer. If the cursor is in the display window, the text scrolls to the start of the source file or program instructions. If the cursor is in the dialog window, the commands scroll to the top of the command buffer. The top of the

command buffer may be blank if you have not yet entered enough commands to fill the buffer. The cursor remains at its current position in the window.

END — Scrolls to the bottom of the file or command buffer.

If the cursor is in the display window, the text scrolls to the end of the source file or program instructions. If the cursor is in the dialog window, the commands scroll to the bottom of the command buffer and the cursor moves to the CodeView prompt (>) at the end of the buffer.

3.2.1.2 Changing the Screen with Keyboard Commands

The following keys change the screen or switch to a different screen:

Key	Function
F1	Displays initial on-line-help screen.
	The help system is discussed in Section 3.2.4. You can also get to the initial help screen by selecting Help from the View menu, as described in Section 3.2.3.3.
F2	Toggles the register window.
	The window disappears if present, or appears if absent. You can also toggle the register window with the Register selection from the Options menu, as described in Section 3.2.3.6.
F3	Switches between source and assembly modes.
	Source mode shows source code in the display window, while assembly mode shows assembly-language instructions. You can also change modes with the Source and Assembly selections from the View menu, as described in Section 3.2.3.3.
F4	Switches to the output screen.
	The output screen shows the output, if any, from your program. Press any key to return to the CodeView screen. You can also change to the output screen with the Output selection from the View menu, as described in Section 3.2.3.3.

3.2.1.3 Controlling Program Execution with Keyboard Commands

The following keys set and clear breakpoints, trace through your program, or execute to a breakpoint:

Key	Function
F5	Executes to the next breakpoint, or to the end of the program if no breakpoint is encountered. This keyboard command corresponds to the Go dialog command when it is given without a destination breakpoint argument.
F7	Sets a temporary breakpoint on the line with the cursor, and executes to that line (or to a previously set breakpoint or the end of the program, if either is encountered before the temporary breakpoint). In source mode, if the line does not correspond to code (for example, data declaration or comment lines), the CodeView debugger sounds a warning and ignores the command. This window command corresponds to the Go dialog command when it is given with a destination breakpoint.
F8	Executes a Trace command. The CodeView debugger executes the next source line in source mode, or the next instruction in assembly mode. If the source line or instruction contains a call to a routine or interrupt, the debugger starts tracing through the call (enters the call and is ready to execute the first source line or instruction). This command will not trace into DOS function calls.
F9	Sets or clears a breakpoint on the line with the cursor. If the line does not currently have a breakpoint, one is set on that line. If the line already has a breakpoint, the breakpoint is cleared. If the cursor is in the dialog window, the CodeView debugger sounds a warning and ignores the command. This window command corresponds to the Breakpoint Set and Breakpoint Clear dialog commands.

F10 — Executes the Program Step command.

The CodeView debugger executes the next source line in source mode, or the next instruction in assembly mode. If the source line or instruction contains a call to a routine or interrupt, the debugger steps over the entire call (executes it to the return) and is ready to execute the line or instruction after the call.

Important

It is usually possible to interrupt program execution by pressing CONTROL-BREAK or CONTROL-C. This can be used to exit endless loops, or it can interrupt loops that are slowed by the Watchpoint or Tracepoint commands (see Chapter 8, "Managing Watch Statements"). CONTROL-BREAK or CONTROL-C may not work if your program has a special use for one or both of these key combinations. If you have an IBM Personal Computer AT (or a compatible computer), you can use the SYSTEM-REQUEST key to interrupt execution regardless of your program's use of CONTROL-BREAK and CONTROL-C.

3.2.1.4 Selecting from Menus with the Keyboard

The CodeView debugger has seven pop-up menus. This section discusses how to make selections from menus. The effects of the selections are discussed in Section 3.2.3.

The menu bar at the top of the screen has nine titles: File, Search, View, Run, Watch, Options, Calls, Trace!, and Go!. The first seven titles are menus, and the last two are commands. The Trace! and Go! titles are provided primarily for mouse users, though you can activate them by pressing ALT-T or ALT-G and then pressing the ENTER key. The exclamation point is a convention used to indicate that a title represents a command rather than a menu. The same commands are more easily accessible with the F5, F8, and F10 keys.

The steps for opening a menu and making a selection are described below:

1. To open a menu, press the ALT key and the first letter of the menu title. For example, press ALT-S to open the Search menu. The menu title is highlighted, and a menu box listing the selections pops up below the title.

2. There are two ways to make a selection from an open menu:

   a. Press the DOWN ARROW key on the numeric keypad to move down the menu. The highlight will follow your movement. When the item you want is highlighted, press the ENTER key to execute the command. For example, press the DOWN ARROW once to select Find from the Search menu.

      You can also press the UP ARROW key to move up the menu. If you move off the top or bottom of the menu, the highlight wraps around to the other end of the menu.

   b. While holding down the ALT key, press the first letter of the item you want to select. You do not have to press the ENTER key with this method. For example, press ALT-F to select Find from the Search menu. This selection method does not work with the Calls menu.

3. One of three things will happen at this point:

   a. For most menu selections, the choice is executed immediately.

   b. The items on the Options menu have small double arrows next to them if the option is on, or no arrows if the option is off. Choosing the item toggles the option. The status of the arrows will be reversed the next time you open the menu.

   c. Some items require a response. In this case, there is another step in the menu-selection process.

4. If the item you select requires a response, a dialog box opens when you select a menu item. Type your response to the prompt in the box and press the ENTER key. For example, the Find dialog box asks you to enter a regular expression (see Appendix B for a complete explanation of regular expressions).

   If your response is valid, the command will be executed. If you enter an invalid response, a message box will appear, telling you the problem and asking you to press a key. Press any key to make the message box disappear.

At any point during the process of selecting a menu item, you can press the ESCAPE key to cancel the menu. While a menu is open, you can press the LEFT ARROW or RIGHT ARROW key to move from one menu to an adjacent menu, or to one of the command titles on the menu bar.

3.2.2 Executing Window Commands with the Mouse

The CodeView debugger is designed to work with the Microsoft Mouse (it also works with some compatible pointing devices). By moving the mouse on a flat surface, you can move the mouse pointer in a corresponding direction on the screen. The following terms refer to the way you select items or execute commands with the mouse:

Term	Definition
Point	To move the mouse until the mouse pointer rests on the item you want to select.
Click	To quickly press and release a mouse button while pointing at an item you want to select.
Drag	To press a mouse button while on a selected item, then hold the button down while moving the mouse. The item moves in the direction of the mouse movement. When the item you are moving is where you want it, release the button; the item will stay at that point.

The CodeView debugger uses two mouse buttons. The terms "click right," "click left," "click both," and "click either" are sometimes used to designate which buttons to use. When dragging, either button can be used.

3.2.2.1 Changing the Screen with the Mouse

You can change various aspects of the screen display by pointing to one of the following elements and then either clicking or dragging:

Item	Action
Double line separating display and dialog windows	Drag the separator line up to increase the size of the dialog window while decreasing the size of the display window, or drag the line down to increase the size of the display window while decreasing the size of the dialog window. You can eliminate either window completely by dragging the line all the way up or down (providing the cursor is not in the window you want to eliminate).

UP ARROW or DOWN ARROW on the scroll bar	Point and click on one of the four arrows on the scroll bars to scroll up or down. If you are in the display window, source code will scroll up or down. If you are in the dialog window, the buffer containing dialog commands entered during the session will scroll up or down. The distance moved is determined by which buttons you click, as follows:

Button	Action
Click left	Scroll up or down, one line at a time
Click right	Scroll up or down, one page at a time; the length of a page is the current size of the window
Click both	Scroll to the top or bottom of the file or command buffer

Some menu selections also change the screen display. See Section 3.2.3 for a description of the menu selections.

3.2.2.2 Controlling Program Execution with the Mouse

Clicking the following mouse items enables you to set and clear breakpoints, trace through your program, execute to a breakpoint, or change the flag bits:

Item	Action
Source line or instruction	Point and click on a source line in source mode or on an instruction in assembly mode to take one of the following actions:

Button	Action
Click left	If the line under the mouse cursor does not have a breakpoint, one is set there. If the line already has a breakpoint, the breakpoint is removed. Lines with breakpoints are shown in high-intensity text.

Click right	A temporary breakpoint is set on the line and the CodeView debugger executes until it reaches the line (or until it reaches a previously set breakpoint or the end of the program if either is encountered before the temporary breakpoint).

If you click on a line that does not correspond to code (for example, a declaration or comment), the CodeView debugger will sound a warning and ignore the command.

Trace! on menu bar

Point and click to trace the next instruction. The kind of trace is determined by the button clicked:

Button	Action
Click left	The Trace command is executed. The CodeView debugger executes the next source line in source mode, or the next instruction in assembly mode. If the source line or instruction contains a call to a routine or interrupt, the debugger starts tracing through the call (it enters the call and is ready to execute the first source line or instruction). This command will not trace into DOS function calls.
Click right	The Program Step command is executed. The debugger executes the next source line in source mode, or the next instruction in assembly mode. If the source line or instruction contains a call to a routine or interrupt, the CodeView debugger steps over the entire call (it executes the call to the return) and is ready to execute the line or instruction after the call.

These two commands are different only if the current location is the start of a procedure, interrupt, or call.

Go! on menu bar	Point and click either button to execute to the next breakpoint, or to the end of the program if no breakpoints are encountered.
Flag in register window	Point to a flag name and click either button to reverse the flag. If the flag bit is set, it will be cleared; if the flag bit is cleared, it will be set. The flag name is changed on the screen to match the new status. If you are using color mode, the color of the flag mnemonic will also change. This command can only be used when the register window is open. Use the command with caution, since changing flag bits can change program execution at the lowest level.

Important

You can usually interrupt program execution by pressing CONTROL-BREAK or CONTROL-C. See the note in Section 3.2.1.3 for more information.

3.2.2.3 Selecting from Menus with the Mouse

The CodeView debugger has seven pop-up menus. This section discusses how to make selections from these menus. The effect of each selection is discussed in Section 3.2.3.

The menu bar at the top of the screen has nine titles: File, Search, View, Run, Watch, Options, Calls, Trace!, and Go!. The first seven titles are menus and the last two are commands that you can execute by clicking with the mouse. The steps for opening a menu and making a selection are described below:

1. To open a menu, point to the title of the menu you want to select. For example, move the pointer onto File on the menu bar if you want to open the File menu.

2. With the mouse pointer on the title, press and hold down either mouse button. The selected title is highlighted and a menu box with a list of selections pops up below the title. For example, if you point to Search and press a button, the Search menu pops up.

3. With the button held down, move the mouse down. The highlight follows the mouse movement. You can move the highlight up or down in the menu box. For example, to select Find from the Search menu, move the highlight down the menu to Find.

   If you move off the box, the highlight will disappear. However, as long as you do not release the button, you can move the pointer back onto the menu to make the highlight reappear.

4. When the selection you want is highlighted, release the mouse button. For example, release the button with the highlight on Find.

   When you release the button, the menu selection is executed. One of three things will happen:

   a. For most menu selections, the choice is executed immediately.

   b. The items on the Options menu have small double arrows next to them if the option is on, or no arrows if the option is off. Choosing the item toggles the option. The status of the arrows will appear reversed the next time you open the menu.

   c. Some items require a response. In this case, there is another step in the menu-selection process.

5. If the item you select requires a response, a dialog box with a prompt appears. Type your response and press the ENTER key or a mouse button. For example, if you selected Find, the prompt will ask you to enter a regular expression (see Section 3.2.3.2, "Using the Search Menu," or Appendix B, "Regular Expressions," for an explanation of regular expressions).

   If your response is valid, the command will be executed. If you enter an invalid response in the dialog box, a message box will appear telling you the problem and asking you to press a key. Press any key or click a mouse button to make the message box disappear.

There are several shortcuts you can take when selecting menu items with the mouse. If you change your mind and decide not to select an item from a menu, just move off the menu and release the mouse button; the menu will disappear. You can move from one menu to another by dragging the pointer directly from any point on the current menu to the title of the new menu.

3.2.3 Using Menu Selections

This section describes the selections on each of the CodeView menus. These selections can be made with the keyboard, as described in Section 3.2.1, or with the mouse, as described in Section 3.2.2.

Note that while the Trace! and Go! commands appear on the menu bar, they are not menus. These titles are provided primarily for mouse users, though you can activate them by pressing ALT-T or ALT-G and then pressing the ENTER key, or by using the F5, F8, and F10 keys.

3.2.3.1 Using the File Menu

The File menu includes selections for working on the current source or program file. The File menu is shown in Figure 3.2, and the selections are explained below:

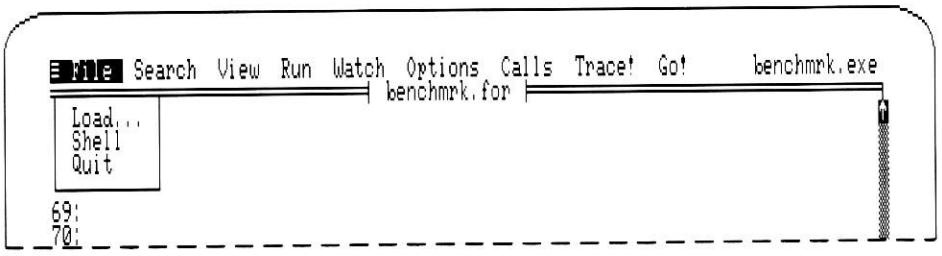

Figure 3.2 The File Menu

Selection	Action
Load...	Opens a new file. When you make this selection, a dialog box appears asking for the name of the new file you want to open. Type the name of a source file, an include file, or any other text file. The text of the new file replaces the current contents of the display window (if you are in assembly mode, the CodeView debugger will switch to source mode). When you finish viewing the file, you can reopen the original source file. The current location and breakpoints will still be marked when you return to the source file.

You do not need to open a new file to see source files for a different module of your program. The CodeView debugger automatically switches to the source file of the other module when program execution enters the other module. While switching source files is never necessary, it may be desirable if you want to set breakpoints or execute to a line in another module.

Note

If the debugger cannot find the source file when it switches modules, a dialog box appears asking for a file specification for the source file. You can either enter a new file specification if the file is in another directory, or press the ENTER key if no source file exists. If you press the ENTER key, the module can only be debugged in assembly mode.

Shell

Exits to a DOS shell. This brings up the DOS screen, where you can execute DOS commands or executable files. To return to the CodeView debugger, type `exit` at the DOS command prompt. The CodeView screen reappears with the same status it had when you left it.

The Shell command works by saving the current processes in memory and then executing a second copy of **COMMAND.COM**. This requires a significant amount of free memory (more than 200K), since the debugger, **COMMAND.COM**, symbol tables, and the debugged program must all be saved in memory. If you do not have enough memory to execute the Shell command, an error message appears. Even if you have enough memory to execute the shell, you may not have enough memory left to execute large programs from the shell.

The Shell command will not work unless you have executed your program's start-up code. You can do this after starting the debugger, or after restarting your program, by executing to any point within the program. For example, enter the dialog command `G main`, or simply press the F10 (Program Step) key.

Quit	Terminates the CodeView debugger and returns to DOS.

3.2.3.2 Using the Search Menu

The Search menu includes selections for searching through text files for text strings and for searching executable code for labels. The Search menu is shown in Figure 3.3 and the selections are explained below:

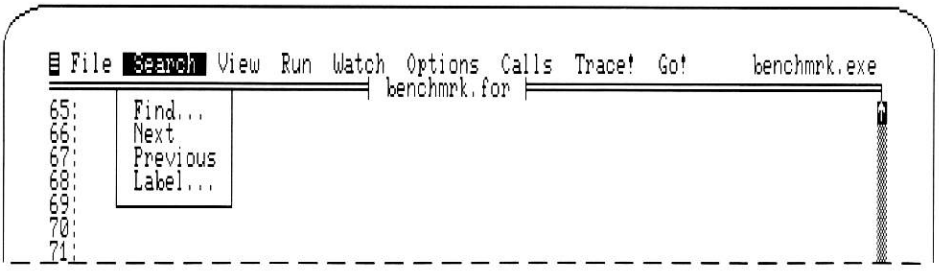

Figure 3.3 The Search Menu

Selection	Action
Find...	Searches the current source file or other text file for a specified regular expression. When you make this selection, a dialog box opens, asking you to enter a regular expression. Type the expression you want to search for and press the ENTER key. The CodeView debugger starts at the current or most recent cursor position in the display window and searches for the expression. If your entry is found, the cursor moves to the first source line containing the expression. If the entry is not found, a message box opens, telling you the problem and asking you to press a key (you can also click a mouse button) to continue. If you are in assembly mode, the debugger automatically switches to source mode when the expression is found.

Regular expressions are a method of specifying variable text strings. This method is similar to the DOS method of using wild cards in file names. Regular expressions are explained in detail in Appendix B.

You can use the Search selections without understanding regular expressions. Since text strings are the simplest form of regular expressions, you can simply enter a string of characters as the expression you want to find. For example, you could enter `count` if you wanted to search for the word "count."

The following characters have a special meaning in regular expressions: backslash (****), asterisk (*****), left bracket ([), period (**.**), dollar sign (**$**), and caret (^). In order to find strings containing these characters, you must precede the characters with a backslash; this cancels their special meanings.

For example, the periods in FORTRAN relational and logical operators must be preceded by backslashes. You would use `\.EQ` to find the **.EQ.** operator.

The Case Sense selection from the Options menu has no effect on searching for regular expressions.

Next

Searches for the next match of the current regular expression.

This selection is meaningful only after you have used the Search command to specify the current regular expression. If the CodeView debugger searches to the end of the file without finding another match for the expression, it wraps around and starts searching at the beginning of the file.

Previous

Searches for the previous match of the current regular expression.

This selection is meaningful only after you have used the Search command to specify the current regular expression. If the debugger searches to the beginning of the file without finding another match for the expression, it wraps around and starts searching at the end of the file.

Label...	Searches the executable code for an assembly-language label. If the label is found, the cursor moves to the instruction containing the label. If you start the search in source mode, the debugger will switch to assembly mode to show a label in a library routine or an assembly-language module.

3.2.3.3 Using the View Menu

The View menu includes selections for switching between source and assembly modes, and for switching among the debugging screen, the output screen, and the help screen. The corresponding function keys for menu selection are shown on the right side of the menu when appropriate. The View menu is shown in Figure 3.4, and the selections are explained below:

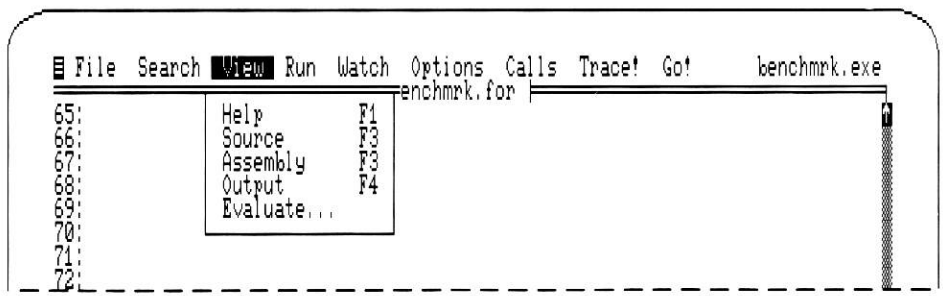

Figure 3.4 The View Menu

Selection	Action
Help	Opens the initial help menu. Section 3.2.4 tells how to move through the on-line-help system and return to the debugging screen.
Source	Changes from assembly mode (showing assembly-language instructions in the display window) to source mode (showing source lines). If you select this mode when you are already in source mode, the selection will be ignored.

Assembly	Changes from source mode (showing source lines in the display window) to assembly mode (showing assembly-language instructions). If you select this mode when you are already in assembly mode, your selection will be ignored.
Output	Replaces the CodeView screen with the output screen. The output screen shows the current output of your program. If the program has not taken over the entire screen, previous DOS commands may still be visible on the output screen. To return to the CodeView screen, press any key or click a mouse button. The output screen is only for viewing; you cannot change it.
Evaluate...	Evaluates a FORTRAN expression. A dialog box opens and asks for the expression to be evaluated. Enter a FORTRAN expression made up of symbols and the FORTRAN operators recognized by the CodeView debugger (see Chapter 4, "Using Dialog Commands"). The value of the expression appears at the CodeView prompt in the dialog window. This selection is similar to the Display Expression dialog command. You can specify the format in which the value will be displayed. Type the expression, followed by a comma and a CodeView format specifier. If you do not give a format specifier, the debugger displays the value in a default format. See Chapter 6, "Examining Data and Expressions," or Appendix A, "Command and Mode Summary," for more information about format specifiers and the default format.

3.2.3.4 Using the Run Menu

The Run menu includes selections for running your program. The Run menu is shown in Figure 3.5, and the selections are explained below:

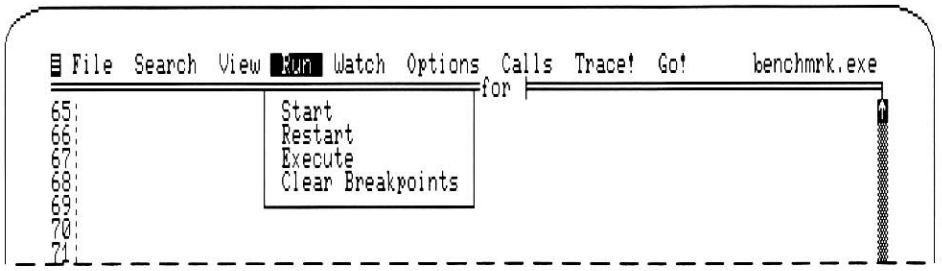

Figure 3.5 The Run Menu

Selection	Action
Start	Starts the program from the beginning and runs it. Any previously set breakpoints or watch statements will still be in effect. The CodeView debugger will run your program from the beginning to the first breakpoint, or to the end of the program if no breakpoint is encountered. This has the same effect as selecting Restart (see the next selection) and then entering the Go command.
Restart	Restarts the current program, but does not begin executing it. You can debug the program again from the beginning. Any previously set breakpoints or watch statements will still be in effect.
Execute	Executes in slow motion from the current instruction. This is the same as the Execute dialog command (**E**). To stop animated execution, press any key or a mouse button.
Clear Breakpoints	Clears all breakpoints. This selection may be convenient after selecting Restart if you don't want to use previously set breakpoints. Note that watch statements are not cleared by this command.

3.2.3.5 Using the Watch Menu

The Watch menu includes selections for managing the watch window. Selections on this menu are also available with dialog commands. The Watch menu is shown in Figure 3.6, and the selections are explained below:

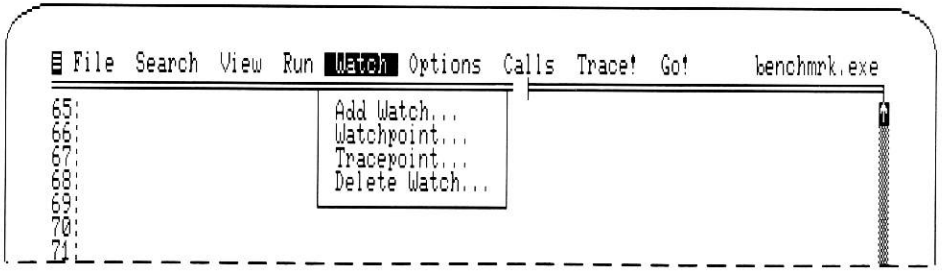

Figure 3.6 The Watch Menu

Selection	Action
Add Watch...	Adds a watch-expression statement to the watch window. A dialog window opens, asking for the FORTRAN expression (which may be simply a variable) whose value you want to see displayed in the watch window. Type the expression and press the ENTER key or press a mouse button. The statement appears in the watch window in normal text. You cannot specify a memory range to be displayed with the Add Watch... selection as you can with the Watch dialog command. You can specify the format in which the value will be displayed. Type the expression, followed by a comma and a CodeView format specifier. If you do not give a format specifier, the CodeView debugger displays the value in a default format. See Chapter 6, "Examining Data and Expressions," or Appendix A, "Command and Mode Summary," for more information about format specifiers and the default format. See Section 8.2, "Setting Watch-Expression and Watch-Memory Statements," for more information about the Watch command.

Watchpoint... Adds a watchpoint statement to the watch window.

A dialog window opens, asking for the FORTRAN expression whose value you want to test. The watchpoint statement appears in the watch window in high-intensity text when you enter the expression. A watchpoint is a conditional breakpoint that causes execution to stop when the expression becomes nonzero (true). See Section 8.3, "Setting Watchpoints," for more information.

Tracepoint... Adds a tracepoint statement to the watch window.

A dialog window opens, asking for the FORTRAN expression or memory range whose value you want to test. The tracepoint statement appears in the watch window in high-intensity text when you enter the expression. A tracepoint is a conditional breakpoint that causes execution to stop when the value of a given expression changes. You cannot specify a memory range to be tested with the Tracepoint... selection as you can with the Tracepoint dialog command.

When setting a tracepoint expression, you can specify the format in which the value will be displayed. Type the expression followed by a comma and a format specifier. If you do not give a format specifier, the CodeView debugger displays the value in a default format. See Chapter 6, "Examining Data and Expressions," for more information about format specifiers and the default format. See Section 8.4, "Setting Tracepoints," for more information about tracepoints.

Delete Watch... Deletes a watch statement from the watch window.

A dialog window opens, showing the current watch statements. If you are using a mouse, move the pointer to the statement you want to delete and click either button. If you are using the keyboard, press the UP ARROW or DOWN ARROW key to move the highlight to the statement you want to delete, then press the ENTER key.

3.2.3.6 Using the Options Menu

The Options menu allows you to set options that affect various aspects of the behavior of the CodeView debugger. The Options menu is shown in Figure 3.7, and the selections are explained below:

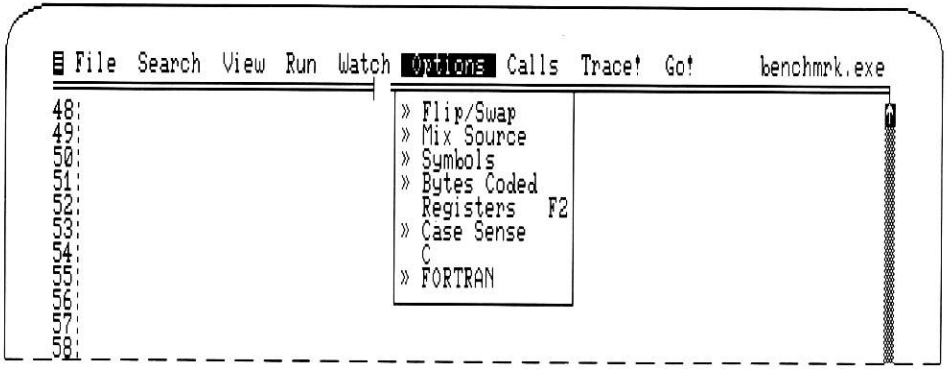

Figure 3.7 The Options Menu

Three of the options control the way assembly-language instructions are displayed in assembly mode. The default setting for assembly mode shows source lines, if available, with the corresponding assembly-language instructions; the bytes for each instruction are shown, as well as the instruction mnemonics; and symbols are used to show variables and labels. The default assembly display for one source line is shown below:

```
12:              READ (*,'(BN,I5)') N
50D9:0079 B8D800          MOV       AX,00D8
50D9:007C 50              PUSH      AX
50D9:007D 9AF5093C53      CALL      __FFrd (533C:09F5)
50D9:0082 83C402          ADD       SP,+02
```

Compare this display with the variations shown in the descriptions for each assembly option.

Selections on the Options menu have small double arrows to the left of the selection name when the option is on. The status of the option (and the presence of the double arrows) is reversed each time you select the option. By default, all the options except Registers and C are on when you start the CodeView debugger with a FORTRAN program.

The selections from the Options menu are discussed below:

Selection	Action
Flip/Swap	When on (the default), screen swapping or screen flipping (whichever the debugger was started with) is active; when off, swapping or flipping is disabled. Turning off swapping or flipping makes the screen scroll more smoothly. You will not see the program flip or swap each time you execute part of the program. This option has no effect if neither swapping nor flipping was selected during start-up.

Warning

Make sure that flipping or swapping is on any time your program writes to the screen. If swapping and flipping are off, your program will write the output at the location of the cursor. The CodeView debugger will detect that the screen has changed and will redraw the screen, thus destroying the program output. The error message `Flip/Swap option off - application output lost` is also displayed.

Mix Source

When on (the default), source lines are shown with assembly-language instructions; when off, instructions are shown without source lines.

This option affects only assembly mode. The sample default display at the beginning of this section shows the appearance of the screen when the option is on. The following display shows the appearance of the same code when the option is off:

```
50D9:0079 B8D800        MOV     AX,00D8
50D9:007C 50            PUSH    AX
50D9:007D 9AF5093C53    CALL    __FFrd (533C:09F5)
50D9:0082 83C402        ADD     SP,+02
```

Symbols

When on (the default), symbols are used to indicate variables; when off, addresses or offsets from register values are shown for variables.

This option affects only assembly mode. The sample default display at the beginning of this section shows the appearance of the screen when the option is on. The following display shows the appearance of the same code when the option is off:

```
12:              READ (*,'(BN,I5)') N
50D9:0079 B8D800          MOV       AX,00D8
50D9:007C 50              PUSH      AX
50D9:007D 9AF5093C53      CALL      533C:09F5
50D9:0082 83C402          ADD       SP,+02
```

Bytes Coded

When on (the default), both the instructions and the bytes for each instruction are shown; when off, only the instructions are shown.

This option affects only assembly mode. The sample default display at the beginning of this section shows the appearance of the screen when the option is on. The following display shows the appearance of the same code when the option is off:

```
12:              READ (*,'(BN,I5)') N
50D9:0079 MOV         AX,00D8
50D9:007C PUSH        AX
50D9:007D CALL        __FFrd (533C:09F5)
50D9:0082 ADD         SP,+02
```

Registers

When on, the register window is displayed; when off (the default), the register window is not displayed.

You can also turn on the register window by pressing F2, as indicated on the right side of the menu.

Case Sense

When on, the CodeView debugger assumes that symbol names are case sensitive (each lowercase letter is different from the corresponding uppercase letter); when off (the default), symbol names are not case sensitive.

Microsoft FORTRAN is normally not case sensitive, so you will probably want to leave this option off. When debugging assembly-language programs, you will probably want to leave it off as well. Note that Case Sense does not affect searches from the Search menu.

C — When on, the CodeView debugger uses the C expression evaluator.

This allows you to debug C program modules using C operators, type casts, and radix specifiers. See CodeView's on-line help for more information about the C expression evaluator.

FORTRAN — When on, the debugger uses the FORTRAN expression evaluator.

This is the default when the main program module is written in FORTRAN. The information in this manual assumes you are using the FORTRAN selection.

Only one expression-evaluator selection can be on at a time. In mixed-language programs, the CodeView debugger automatically selects the correct expression evaluator. Note that using the expression-evaluator selections to override the current language (for example, C when debugging a FORTRAN module) will cause unpredictable behavior, mostly in the form of unrecognized variables and operators.

3.2.3.7 Using the Calls Menu

The Calls menu is different from other menus in that its contents and size change, depending on the status of your program. The Calls menu is shown in Figure 3.8.

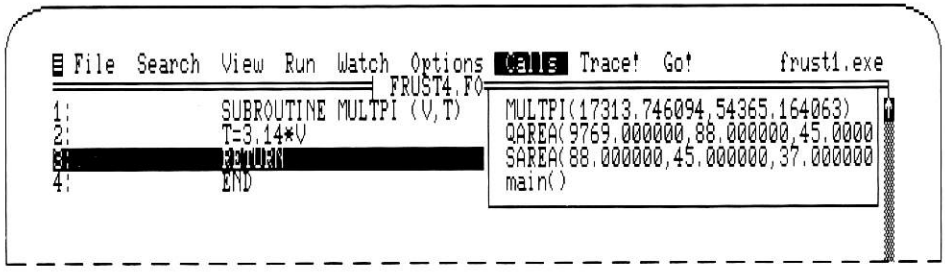

Figure 3.8 The Calls Menu

Like other menus, the Calls menu can be opened by pressing the ALT key and the first letter of the menu title (ALT-C). However, you cannot make selections from the Calls menu by pressing the ALT key with the first letter of your selection. You must use the UP ARROW or DOWN ARROW key to move to your selection, then press the ENTER key. You can also use the mouse to open and select from the Calls menu. Usually the menu is used to view the current functions rather than to actually make selections.

The Calls menu shows the current routine and the trail of routines from which it was called. The current routine is always at the top. The routine from which the current routine was called is directly below. Other active routines are shown in the reverse order in which they were called. With FORTRAN programs, the bottom routine will always be **main**.

The current value of each argument, if any, is shown in parentheses following the routine. The menu expands to accommodate the arguments of the widest routine. Arguments are shown in the current radix (the default is decimal). If there are more active routines than will fit on the screen, or if the routine arguments are too wide, the display will be truncated. The Stack Trace dialog command (**K**) shows all the routines and arguments regardless of the size of the display.

If you want to view code at the point where one of the routines was called, select the routine below the one you want to view. The cursor will move to the calling source line (in source mode) or to the calling instruction (in assembly mode). In other words, the cursor will indicate the calling location in the selected routine where the next-level routine was called. If you select the current (top-level) routine, the cursor moves to the current location in that routine.

For example, if the Calls menu shown in Figure 3.8 appears and you select SAREA, the cursor will move to the line in main where the routine SAREA is called.

Note

If you are using the CodeView debugger to debug assembly-language programs, routines will be shown in the Calls menu only if they use the Microsoft calling convention. This calling convention is explained in Chapter 11 of the *Microsoft FORTRAN Compiler User's Guide*.

3.2.4 Using the Help System

The CodeView on-line-help system uses tree-structured menus to give you quick access to help screens on a variety of subjects. The system is organized by subject so that you can reach a help screen on any topic with a minimum of keystrokes.

The help file is called **CV.HLP**. It must be present in the current directory or in one of the directories specified with the DOS **PATH** command. If the help file is not found, the CodeView debugger will still operate, but you will not be able to use the help system. An error message will appear if you try to use a help command.

When you request help, either by pressing the F1 key or by selecting Help from the View menu, the top-level help menu appears. You select one of the topics listed by pressing the highlighted first letter of the menu title. You can also select a topic by pointing to it with the mouse and clicking either button.

When you select a topic, the help screen on that subject may appear immediately, or in some cases, a second menu screen will appear. Keep selecting topics until you reach the screen you want.

In addition to menu titles, you can select four special commands from any help screen. The keys that select these commands are always shown at the top of the screen. You can also point and click with the mouse to select these commands. The commands are listed below:

Command	Action
PGUP	Returns to the previous help screen or menu.
PGDN	Proceeds to the next-level screen. This command is intended for help topics that consist of more than one screen of text. If you press this key on a menu screen, the top leftmost selection is chosen as the default. The CodeView debugger sounds a warning if there is no lower-level screen.
HOME	Returns to the top-level menu. The debugger sounds a warning if you are already at the top level.

END	Returns to the debugging screen.
ESC	Returns to the debugging screen.

When you use a selection letter to select a topic that has more than one screen of information, you can press the selection letter again to get the next screen. This has the same effect as pressing the PGDN key.

3.3 Using Sequential Mode

Sequential mode is required if you have neither an IBM Personal Computer nor a closely compatible computer. In sequential mode, the CodeView debugger works much like its predecessors, the Microsoft Symbolic Debug Utility (**SYMDEB**) and the DOS **DEBUG** utility.

In sequential mode, the CodeView debugger's input and output always move down the screen from the current location. When the screen is full, the old output scrolls off the top of the screen to make room for new output appearing at the bottom. You can never return to examine previous commands once they scroll off, but in many cases, you can reenter the command to put the same information on the screen again.

Most window commands cannot be used in sequential mode. However, the following function keys, which are used as commands in window mode, are also available in sequential mode:

Command	Action
F1	Displays a command-syntax summary. This is equivalent to the Help (**H**) dialog command. It is different from the on-line-help system accessed by the F1 key and the Help dialog command in window mode.
F2	Displays the registers. This is equivalent to the Register (**R**) dialog command.
F3	Toggles between source and assembly modes. If the current mode is source (displaying source lines as output to the Trace, Program Step, and Go commands), the system switches to assembly mode (displaying

assembly-language instructions). If the current mode is assembly, the system switches to source. This is equivalent to using the Set Assembly (**S−**) and Set Source (**S+**) dialog commands.

F4 — Switches to the output screen, which shows the output of your program.

Press any key to return to the CodeView debugging screen. This is equivalent to the Screen Exchange (****) dialog command.

F5 — Executes from the current instruction until a breakpoint or the end of the program is encountered.

This is equivalent to the Go dialog command (**G**) with no argument.

F8 — Executes the next source line in source mode, or the next instruction in assembly mode.

If the source line or instruction contains a function, procedure, or interrupt call, the CodeView debugger executes the first source line or instruction of the call and is ready to execute the next source line or instruction within the call. This is equivalent to the Trace (**T**) dialog command.

F9 — Sets or clears a breakpoint at the current program location.

If the current program location has no breakpoint, one is set. If the current location has a breakpoint, it is removed. This is equivalent to the Breakpoint Set (**BP**) dialog command with no argument.

F10 — Executes the next source line in source mode, or the next instruction in assembly mode.

If the source line or instruction contains a function, procedure, or interrupt call, the call is executed to the end and the CodeView debugger is ready to execute the line or instruction after the call. This is equivalent to the Program Step (**P**) dialog command.

The CodeView Watch (**W**), Watchpoint (**WP**), and Tracepoint (**TP**) commands work in sequential mode, but since there is no watch window, the watch statements are not shown. You must use the Watch List command (**W**) to examine watch statements and watch values. See Chapter 8 for information on Watch Statement commands.

All the CodeView commands that affect program operation (such as Trace, Go, and Breakpoint Set) are available in sequential mode. Any debugging operation that can be done in window mode can also be done in sequential mode.

Chapter 4
Using Dialog Commands

4.1 Introduction 71
4.2 Entering Commands and Arguments 71
4.2.1 Using Special Keys 71
4.2.2 Using the Command Buffer 72
4.3 Format for CodeView Commands and Arguments 73
4.4 Expressions 74
4.4.1 Symbols 78
4.4.2 Constant Numbers 78
4.4.3 Registers 80
4.4.4 Addresses 81
4.4.5 Address Ranges 82
4.4.6 Line Numbers 83
4.4.7 Strings 84
4.4.8 Switching Expression Evaluators 84

4.1 Introduction

CodeView dialog commands can be used in sequential mode or from the dialog window. In sequential mode, they are the primary method of entering commands. In window mode, dialog commands are used to enter commands that require arguments or do not have corresponding window commands.

Many window commands have duplicate dialog commands. Generally, the window version of a command is more convenient, while the dialog version is more powerful. For example, to set a breakpoint on a source line in window mode, put the cursor on the source line and press F9, or point to the line and click the left mouse button. The dialog version of the command (**BP**) requires more keystrokes, but it allows you to specify an address, a pass count, and a string of commands to be taken whenever the breakpoint is encountered.

The rest of this chapter explains how to enter dialog commands and how to specify command arguments.

4.2 Entering Commands and Arguments

Dialog commands are entered at the CodeView prompt (>). Type the command and arguments, then press the ENTER key.

In window mode, you can enter commands regardless of whether the cursor is at the CodeView prompt. If the cursor is in the display window, the text you type will appear after the prompt in the dialog window, even though the cursor remains in the display window.

4.2.1 Using Special Keys

While entering dialog commands or viewing output from commands, you can use the following special keys:

Key	Action
CONTROL-C	Stops the current output or cancels the current command line.

For example, if you are watching a long display from a Dump command, you can press CONTROL-C to interrupt the output and return to the CodeView prompt. If you make a mistake while entering a command, you can press CONTROL-C to cancel the command without executing it. A new prompt appears and you can reenter the command.

CONTROL-S — Pauses during output of a command.

You can press any key to continue output. For example, if you are watching a long display from a Dump command, you can press CONTROL-S when a part of the display that you want to examine more closely appears. Then press any key when you are ready for the output to continue scrolling.

Note that CONTROL-S does not work on machines that are not IBM compatible, since the CodeView debugger does not perform its output through DOS.

BACKSPACE — Deletes the previous character on the command line and moves the cursor back one space.

For example, if you make an error while typing a command, you can use the BACKSPACE key to delete the characters back to the error, then retype the rest of the command.

4.2.2 Using the Command Buffer

In window mode, the CodeView debugger has a command buffer where the last 4K (4096 bytes) of commands and command output are stored. This amounts to approximately three screens of text, depending on the length of your commands and output. The command buffer is not available in sequential mode.

When the cursor is in the dialog window, you can scroll up or down to view the commands you have entered earlier in the session. The commands for moving the cursor and scrolling through the buffer are explained in Sections 3.2.1.1 and 3.2.2.1.

Scrolling through the buffer is particularly useful for viewing the output from commands, such as Dump or Examine Symbols, whose output may scroll off the top of the dialog window.

If you have scrolled through the dialog buffer to look at previous commands and output, you can still enter new commands. When you type a command, it will appear to be overwriting the previous line where the cursor is located, but when you press the ENTER key, the new command will be entered at the end of the buffer. For example, if you enter a command while the cursor is at the start of the buffer, and then scroll to the end of the buffer, you will see the command you just entered. If you scroll back to the point where you entered the command, you will see the original characters rather than the characters you typed over the originals.

When you start the debugger, the buffer is empty except for the copyright message. As you enter commands during the session, the buffer is gradually filled from the bottom to the top. If you have not filled the entire buffer and you press the HOME key to go to the top of the buffer, you will not see the first commands of the session. Instead you will see blank lines, since there is nothing at the top of the buffer.

4.3 Format for CodeView Commands and Arguments

The CodeView command format is similar to the format of previous Microsoft debuggers, **SYMDEB** and **DEBUG**. However, some features, particularly operators and expressions, are different. The general format for CodeView commands is shown below:

command [[*arguments*]] [[;*command2*]]

The *command* is a one-, two-, or three-character command name. Any combination of uppercase and lowercase letters can be used in commands. The *arguments* are expressions that represent values or addresses to be used by the command. You may state an expression at either the assembly-language level or the FORTRAN source level. Arguments are normally not case sensitive (unless you turn case sensitivity on with the Case Sense selection from the Options menu). Usually, the first *argument* can be placed immediately after *command* with no space separating the two.

The number of arguments required or allowed with each command varies. If a command takes two or more arguments, you must separate the arguments with spaces. A semicolon (;) can be used as a command separator if you want to specify more than one command on a line.

■ Examples

```
>DB #100 #200
```

In the above example, DB is the first command (for the Dump Bytes command). The arguments to the command are the hexadecimal addresses #100 and #200.

```
>? XYZ
```

In the above example, ? is the command character (for the Display Expression command) and XYZ is the command argument, a FORTRAN variable.

```
>U label1 ;DB
```

The above example consists of two commands, separated by semicolons. The first is the Unassemble command (U) with label1, an assembly-language label, as an argument. The second is the Dump Bytes command (DB) with no arguments.

4.4 Expressions

CodeView arguments are expressions that can include symbols, constant numbers, operators, and registers. Arguments may be simple assembly-language expressions that directly specify an address or range in memory, or they may be FORTRAN expressions employing any of the FORTRAN operators and intrinsic functions supported by the CodeView debugger. FORTRAN function calls are also permitted. Statement function names and COMMON block names are not allowed in expressions. While subroutine names are allowed in expressions, their use is not recommended, because the value returned by the expression will be meaningless.

The CodeView debugger uses a subset of operators consisting of the most commonly used FORTRAN operators and two additional operators, the period (.) and colon (:). Note that you are limited to these operators only in expressions that *you* input to the debugger. The program you are debugging may contain any valid FORTRAN expression. The CodeView operators are listed in Table 4.1 in order of precedence.

Table 4.1
CodeView Operators

Precedence	Operator
(Highest)	
1	()
2	. :
3	Unary + −
4	* /
5	Binary + −
6	.LT. .LE. .EQ. .NE. .GT. .GE.
7	.NOT.
8	.AND.
9	.OR.
10	.EQV. .NEQV.
11	=
(Lowest)	

The CodeView debugger does not support the character concatenation operator (**//**) or the exponentiation operator (******).

The order and precedence with which the CodeView debugger evaluates FORTRAN expressions is the same as in the Microsoft FORTRAN language. See Chapter 2 of the *Microsoft FORTRAN Compiler Language Reference* for a description of how FORTRAN operators can be combined with symbols and constants to form expressions.

The colon operator (**:**) may be used when specifying a memory address. It acts as a *segment*:*offset* separator, as described in Section 4.4.3, "Registers."

In the CodeView debugger, the period (**.**) has an extended use as a specifier of local variables in parent routines. The syntax is shown below:

routine.variable

The *routine* must be a higher-level routine and the *variable* must be a local variable within the specified routine. For example, you can use the expression `main.X` to refer to the local variable `X` in the procedure `main` if you are in a routine that is called by `main`. Note that in this example, `main` refers to the main routine of the FORTRAN program being debugged, and is a symbol generated by the Microsoft FORTRAN Optimizing Compiler. It does not appear in FORTRAN source code.

Note

When you try to use an undefined variable in an expression, the CodeView debugger displays the message `Unknown symbol`. For example, the message appears if you attempt to reference a variable before the program you are debugging has started or after it has terminated, or if you try to reference a local variable outside the routine where the variable is defined.

When entering a FORTRAN expression, you may use a limited number of FORTRAN intrinsic functions. The primary use of these functions is to convert a FORTRAN variable or value from one type to another for purposes of calculation. The intrinsic functions recognized by the CodeView debugger's expression evaluator are listed in Table 4.2. See Chapter 3 of the *Microsoft FORTRAN Compiler Language Reference* for a complete description of the FORTRAN intrinsic functions.

Table 4.2

FORTRAN Intrinsic Functions Supported by the CodeView Debugger

Name	Definition	Argument Type	Function Type
CHAR(*int*)	Data-type conversion	int	char[1]
CMPLX(*genA*[,*genB*])	Data-type conversion	int, real, or cmp	cmp8
DBLE(*gen*)	Data-type conversion	int, real, or cmp	dbl
DCMPLX(*genA*[,*genB*])	Data-type conversion	int, real, or cmp	cmp16
DIMAG(*cmp16*)	Imaginary part of cmp16 number	cmp16	dbl
DREAL(*cmp16*)	Data-type conversion	cmp16	dbl
ICHAR(*char*)	Data-type conversion	char	int
IMAG(*cmp*)	Imaginary part of cmp number	cmp	real[2]
INT(*gen*)	Data-type conversion	int, real, or cmp	int
INT1(*gen*)	Data-type conversion	int, real, or cmp	int1
INT4(*gen*)	Data-type conversion	int, real, or cmp	int4
INTC(*gen*)	Data-type conversion	int, real, or cmp	**INTEGER**[C]
LOCFAR(*gen*)	Segmented address	int, real, or cmp	int4
LOCNEAR(*gen*)	Unsegmented address	int, real, or cmp	int2
REAL(*gen*)	Data-type conversion	int, real, or cmp	real4

[1] The abbreviations used for the different data types in this table are listed in Appendix B of the *Microsoft FORTRAN Compiler Language Reference*.

[2] If argument is **COMPLEX*8**, function is **REAL*4**. If argument is **COMPLEX*16**, function is **DOUBLE PRECISION**.

Sections 4.3 – 4.4 describe how to specify arguments using CodeView expressions.

4.4.1 Symbols

■ Syntax

name

A symbol is a name that represents a register, an absolute value, a segment address, or an offset address. At the FORTRAN source level, a symbol is a variable name or the name of a routine. Note that when given as arguments, symbols are never case sensitive in the FORTRAN version of the CodeView debugger. If you have turned on case sensitivity with the Case Sense selection from the Options menu, it is turned off automatically when a symbol is used.

In assembly-language output or in output from the Examine Symbols command, the CodeView debugger displays some symbol names in the object-code format produced by the Microsoft FORTRAN Optimizing Compiler. This format includes a leading underscore. For example, the main routine in your program will be displayed as `_main`. Only global labels (such as procedure names) are shown in this format. You do not need to include the underscore when specifying such a symbol in CodeView commands. Labels within library routines are sometimes displayed with a double underscore (`__chkstk`). You must use leading underscores when accessing these labels with CodeView commands.

4.4.2 Constant Numbers

■ Syntax

digits	Default radix
radix#*digits*	Specified radix
#*digits*	Hexadecimal

Numbers used in CodeView commands represent integer constants. They are entered in the current input radix (base). When you are using the FORTRAN expression evaluator, the debugger will recognize any radix between 2 and 36, inclusive, that is explicitly specified, as in `20#2G`. (If you switch to the C expression evaluator, the CodeView debugger will recognize only

octal, decimal, and hexadecimal radixes.) The FORTRAN radix specifiers can be used to override the current radix. Note that a hexadecimal number may be entered in two ways. For example, 3F hex could be entered as either `#3F` or `16#3F`. In this manual, we will use the number sign alone to indicate hexadecimal numbers.

The default radix for the FORTRAN version of the CodeView debugger is decimal. However, you can use the Radix command (**N**) to specify an octal or hexadecimal radix, as explained in Section 11.4, "Radix Command."

With radix 16, it is possible to enter a value or argument that could be interpreted either as an identifier or as a hexadecimal number. The CodeView debugger resolves the ambiguity by searching first for a symbol (identifier) with that name. If no symbol is found, the debugger interprets the value as a hexadecimal number. If you want to enter a number that overrides an existing symbol, use the hexadecimal format (*#digits*).

For example, if you enter `ABC` as an argument when the program contains a variable or function named `ABC`, the CodeView debugger interprets the argument as the symbol. If you want to enter `ABC` as a number, enter it as `#ABC`.

Table 4.3 shows how a sample number (63 decimal) would be represented in the octal, decimal, and hexadecimal radixes.

Table 4.3

CodeView Radix Examples

Input Radix	Octal	Decimal	Hexadecimal
8	77	10#63	#3F
10	8#77	63	#3F
16	8#77	10#63	3F

4.4.3 Registers

■ Syntax

[[@]]*register*

You can specify a register name if you want to use the current value stored in the register. Registers are rarely needed in FORTRAN source debugging, but they are used frequently for assembly-language debugging.

When you specify an identifier, the CodeView debugger first checks the symbol table for a symbol with that name. If the debugger does not find a symbol, it checks to see if the identifier is a valid register name. If you want the identifier to be considered a register, regardless of any name in the symbol table, use the "at sign" (@) as a prefix to the register name. For example, if your program has a symbol called `AX`, you could specify `@AX` to refer to the **AX** register. You can avoid this problem entirely by making sure that identifier names in your program do not conflict with register names.

The register names recognized by the CodeView debugger are shown in Table 4.4.

Table 4.4

Registers

Type	Names			
16-bit general purpose	**AX**	**BX**	**CX**	**DX**
16-bit segment	**CS**	**DS**	**SS**	**ES**
16-bit pointer	**SP**	**BP**	**IP**	
16-bit index	**SI**	**DI**		

4.4.4 Addresses

■ Syntax

[[*segment*:]]*offset*

Addresses can be specified in the CodeView debugger through use of the colon operator as a *segment*:*offset* connector. Both the *segment* and the *offset* are made up of expressions.

A full address has a *segment* and an *offset*, separated by a colon. A partial address has just an *offset*, and a default segment is assumed. The default *segment* varies, depending on the command with which the address is used. Commands that refer to data (Dump, Enter, Watch, and Tracepoint) use the contents of the **DS** register. Commands that refer to code (Assemble, Breakpoint Set, Go, Unassemble, and View) use the contents of the **CS** register.

Full addresses are seldom necessary in source-level debugging. Occasionally they may be convenient for referring to addresses outside the program, such as BIOS (basic input/output system) or DOS addresses.

■ Examples

```
>DB 100
```

In the above example, the Dump Bytes command (**DB**) is used to dump memory starting at offset address `100`. Since no segment is given, the data segment (the default for Dump commands) is assumed.

```
>DB ARRAY(COUNT)
```

In the above example, the Dump Bytes command is used to dump memory starting at the address of the variable `ARRAY(COUNT)`.

```
>DB label+10
```

In the above example, the Dump Bytes command is used to dump memory starting at a point 10 bytes beyond the symbol `label`.

```
>DB #B800:#FF
```

In the above example, the Dump Bytes command is used to dump memory at the absolute address having the segment value #B800 and the offset address #FF.

4.4.5 Address Ranges

■ Syntax

startaddress endaddress
startaddress **L** *count*

A range is a pair of memory addresses that bound a sequence of contiguous memory locations.

You can specify a range in two ways. One way is to give the start and end points. In this case the range covers *startaddress* to *endaddress*, inclusive. If a command takes a range, but you do not supply a second address, the CodeView debugger usually assumes the default range. Each command has its own default range. (The most common default range is 128 bytes.)

You can also specify a range by giving its starting point and the number of objects you want included in the range. This type of range is called an object range. In specifying an object range, *startaddress* is the address of the first object in the list, **L** indicates that this is an object range rather than an ordinary range, and *count* specifies the number of objects in the range.

The size of the objects is the size taken by the command. For example, the Dump Bytes command (**DB**) has byte objects, the Dump Words command (**DW**) has words, the Unassemble (**U**) command has instructions, and so on.

■ Examples

```
>DB buffer
```

The above example dumps a range of memory starting at the symbol `buffer`. Since the end of the range is not given, the default size (128 bytes for the Dump Bytes command) is assumed.

```
>DB buffer buffer+20
```

The above example dumps a range of memory starting at `buffer` and ending at `buffer+20` (the point 20 bytes beyond `buffer`).

```
>DB buffer L 20
```

The above example uses an object range to dump the same range as in the previous example. The `L` indicates that the range is an object range, and `20` is the number of objects in the range. Each object has a size of 1 byte, since that is the command size.

```
>U funcname-30 funcname
```

The above example uses the Unassemble command (`U`) to list the assembly-language statements starting 30 instructions before `funcname` and continuing to `funcname`.

4.4.6 Line Numbers

■ Syntax

.[[*filename*:]]*linenumber*

The address corresponding to a source line number can be specified as *linenumber* prefixed with a period (.). The CodeView debugger assumes the source line is in the current source file unless you specify the optional *filename* followed by a colon and the line number.

The CodeView debugger displays an error message if *filename* does not exist, or if no source line exists for the specified number.

■ Examples

```
>V .100
```

The above example uses the View command (**V**) to display code starting at the address that corresponds to source line `100`. Since no file is indicated, the current source file is assumed.

```
>V .SAMPLE.FOR:10
```

The example shown above uses the View command to display source code starting at the address that corresponds to source line 10 of the source file SAMPLE.FOR.

4.4.7 Strings

■ Syntax

'*string*'

Strings can be specified as character expressions in the FORTRAN format. Single quotation marks within a string must be specified by two single quotation marks.

■ Example

```
>EA MESSAGE 'This ''string'' is okay.  '
```

The above example uses the Enter ASCII command (**EA**) to enter the given string into memory starting at the address of the variable MESSAGE. Note that the string includes embedded single quotation marks and trailing blanks.

4.4.8 Switching Expression Evaluators

The CodeView debugger uses the program's source-file extension to automatically select the appropriate "expression evaluator" to use when evaluating expressions that appear in the program or that you enter as dialog commands. If the file extension is not **.C** or **.FOR**, the debugger uses the C expression evaluator.

The CodeView debugger also allows you to specify a particular expression evaluator: FORTRAN or C. Specifying a language is useful when you want to use features of another language to look at your program.

Note

To debug program modules at the source level you must compile and link using the correct versions of the compiler and linker. See your language documentation for information about compiling and linking programs for use with the CodeView debugger.

■ Mouse

To switch expression evaluators with the mouse, open the Options menu and click the appropriate language selection.

■ Keyboard

There is no keyboard equivalent to this command.

■ Dialog

To switch expression evaluators using a dialog command, enter a command line with the syntax

USE [[*language*]]

where *language* is C or FORTRAN. The command is not case sensitive, and you can enter the language name in any combination of uppercase and lowercase letters. Entered on a line by itself, **USE** displays the name of the current expression evaluator.

■ Examples

```
>USE fortran
>FORTRAN
```

The above example switches to the FORTRAN expression evaluator.

```
>USE
FORTRAN
```

The above example displays the name of the current expression evaluator.

Chapter 5
Executing Code

5.1 Introduction 89
5.2 Trace Command 90
5.3 Program Step Command 93
5.4 Go Command 95
5.5 Execute Command 98
5.6 Restart Command 99

5.1 Introduction

Several commands execute code within a program. Among the differences between them is the size of step executed by each command. The commands and their step sizes are listed below:

Command	Action
Trace (**T**)	Executes the current source line in source mode, or the current instruction in assembly mode; traces into routines, procedures, or interrupts
Program Step (**P**)	Executes the current source line in source mode, or the current instruction in assembly mode; steps over routines, procedures, or interrupts
Go (**G**)	Executes the current program
Execute (**E**)	Executes the current program in slow motion
Restart (**L**)	Restarts the current program

In window mode, the screen is updated to reflect changes that occur when you execute a Trace, Program Step, or Go command. The highlight marking the current location is moved to the new current instruction in the display window. Values are changed, if appropriate, in the register and watch windows.

In sequential mode, the current source line or instruction is displayed after each Trace, Program Step, or Go command. The format of the display depends on the display mode. The three display modes available in sequential mode (source, assembly, and mixed) are discussed in Chapter 9, "Examining Code."

If the display mode is source (**S+**) in sequential mode, the current source line is shown. If the display mode is assembly (**S−**), the status of the registers and flags and the new current instruction are shown in the format of the Register command (see Chapter 6, "Examining Data and Expressions"). If the display mode is mixed (**S&**), the registers, the new source line, and the new instruction are all shown.

The commands that execute code are explained in Sections 5.2 – 5.6.

Note

If you are executing a section of code with the Go or Program Step command, you can usually interrupt program execution by pressing CONTROL-BREAK or CONTROL-C. This can terminate endless loops, or it can interrupt loops that are delayed by the Watchpoint or Tracepoint command (see Chapter 8, "Managing Watch Statements"). CONTROL-BREAK or CONTROL-C may not work if your program has a special use for either of these key combinations. If you have an IBM Personal Computer AT (or a compatible computer), you can use the SYSTEM-REQUEST key to interrupt execution regardless of your program's use of CONTROL-BREAK and CONTROL-C.

5.2 Trace Command

The Trace command executes the current source line in source mode, or the current instruction in assembly mode. The current source line or instruction is the one pointed to by the **CS** and **IP** registers. In window mode, the current instruction is shown in reverse video or in a contrasting color.

In source mode, if the current source line contains a call, the CodeView debugger executes the first source line of the called routine. In assembly mode, if the current instruction is **CALL**, **INT**, or **REP**, the debugger executes the first instruction of the procedure, interrupt, or repeated string sequence.

Use the Trace command if you want to trace into calls. If you want to execute calls without tracing into them, you should use the Program Step command (**P**) instead. Both commands execute *DOS* function calls without tracing into them. There is no direct way to trace into *DOS* function calls.

In source mode, the CodeView debugger will only trace into functions and routines that have source code. For example, if the current line contains a call to the intrinsic function **SIN**, the debugger will execute the function if you are in source mode, since the source code for the FORTRAN library is not available. If you are in assembly or mixed mode, the debugger will trace into the function.

Note

The Trace command (**T**) uses the hardware trace mode of the 8086 family of processors. Consequently, you can also trace instructions stored in ROM (read-only memory). However, the Program Step command (**P**) will not work in ROM. Using the Program Step command has the same effect as using the Go command (**G**).

■ Mouse

To execute the Trace command with the mouse, point to Trace on the menu bar and click the left button.

■ Keyboard

To execute the Trace command with a keyboard command, press the F8 key. This works in both window and sequential modes.

■ Dialog

To execute the Trace command using a dialog command, enter a command line with the following syntax:

T [[*count*]]

If the optional *count* is specified, the command executes *count* times before stopping.

■ Examples

The following examples all show the Trace command in sequential mode. In window mode, there would be no output from the commands, but the display would be updated to show changes caused by the command.

```
>S+
source
>.
9:                CALL INPUT (DATA,N,INPFMT)
>T 3
34:               OPEN (1,FILE='EXAMPLE.DAT',STATUS='OLD')
35:               DO 100 I=1,N
36:               READ (1,'(BN,I10)',END=999) DATA(I)

>
```

The above example sets the display mode to source, then uses the Source Line command to display the current source line. (See Chapter 9, "Examining Code," for a further explanation of the Set Source and Source Line commands.) Note that the current source line calls the subprogram INPUT. The Trace command is then used to execute the next three source lines. These lines will be the first three lines of the subprogram INPUT.

```
>S-
assembly
>T
AX=0058  BX=3050  CX=000B  DX=3FB0  SP=304C  BP=3056  SI=00CC  DI=40E0
DS=49B7  ES=49B7  SS=49B7  CS=3FB0  IP=0013  NV UP EI PL NZ AC PO NC
3FB0:0013 50              PUSH      AX
>
```

The above example sets the display mode to assembly and traces the current instruction. This example and the next example are the same as the examples of the Program Step command in Section 5.3. The Trace and Program Step commands behave differently only when the current instruction is a call.

```
>S&
mixed
>T
AX=0000  BX=319C  CX=0028  DX=0000  SP=304C  BP=3056  SI=00CC  DI=40E0
DS=49B7  ES=49B7  SS=49B7  CS=3FB0  IP=003C  NV UP EI PL NZ NA PO NC
8:                  IF (N.LT.1 .OR. N.GT.1000) GO TO 100
3FB0:003C 833ECE2101   CMP     Word Ptr [21CE],+01        DS:21CE=0028
>
```

The above example sets the display mode to mixed and traces the current instruction.

5.3 Program Step Command

The Program Step command executes the current source line in source mode, or the current instruction in assembly mode. The current source line or instruction is the one pointed to by the **CS** and **IP** registers. In window mode, the current instruction is shown in reverse video or in a contrasting color.

In source mode, if the current source line contains a call, the CodeView debugger executes the entire routine and is ready to execute the line after the call. In assembly mode, if the current instruction is **CALL**, **INT**, or **REP**, the debugger executes the entire procedure, interrupt, or repeated string sequence.

Use the Program Step command if you want to execute over routine, function, procedure, and interrupt calls. If you want to trace into calls, you should use the Trace command (**T**) instead. Both commands execute DOS function calls without tracing into them. There is no direct way to trace into DOS function calls.

■ Mouse

To execute the Program Step command with the mouse, point to Trace on the menu bar and click the right button.

■ Keyboard

To execute the Program Step command with a keyboard command, press the F10 key. This works in both window and sequential modes.

■ Dialog

To execute the Program Step command using a dialog command, enter a command line with the following syntax:

P [[*count*]]

If the optional *count* is specified, the command executes *count* times before stopping.

■ Examples

The examples show the Program Step command in sequential mode. In window mode, there would be no output from the commands, but the display would be updated to show changes caused by the command.

```
>S+
source
>.
9:                CALL  INPUT (DATA,N,INPFMT)
>P 3
10:               CALL  BUBBLE (DATA,N)
11:               CALL  STATS (DATA,N)
12:               END
>
```

The above example sets the display mode to source, then uses the Source Line command to display the current source line. (See Chapter 9, "Examining Code," for a further explanation of the Set Source and Source Line commands.) Note that the current source line calls the subprogram INPUT. The Program Step command is then used to execute the next three source lines. The first program step executes the entire subprogram INPUT. The next two steps execute the subprograms BUBBLE and STATS, also in their entirety.

```
>S-
assembly
>P
AX=0058  BX=3050  CX=000B  DX=3FB0  SP=304C  BP=3056  SI=00CC  DI=40E0
DS=49B7  ES=49B7  SS=49B7  CS=3FB0  IP=0013  NV UP EI PL NZ AC PO NC
3FB0:0013 50              PUSH      AX
>
```

The example shown above sets the display mode to assembly and steps through the current instruction. This example and the next example are the same as the examples of the Trace command in Section 5.2. The Trace and Program Step commands behave differently only when the current instruction is a call.

```
>S&
mixed
>P
AX=0000  BX=319C  CX=0028  DX=0000  SP=304C  BP=3056  SI=00CC  DI=40E0
DS=49B7  ES=49B7  SS=49B7  CS=3FB0  IP=003C  NV UP EI PL NZ NA PO NC
8:                  IF (N.LT.1 .OR. N.GT.1000) GO TO 100
3FB0:003C 833ECE2101    CMP     Word Ptr [21CE],+01          DS:21CE=0028
>
```

The above example sets the display mode to mixed and steps through the current instruction.

5.4 Go Command

The Go command starts execution at the current address. There are two variations of the Go command. One simply starts execution and continues to the end of the program or until a breakpoint is encountered. The other variation is a Goto command, in which a destination is given with the command.

The Go command will stop execution when a breakpoint set earlier with the Breakpoint Set (**BP**), Watchpoint (**WP**), or Tracepoint (**TP**) command is encountered. If the command is given in the Goto form, the execution will stop before the destination is reached if a previously set breakpoint is encountered first.

If a destination address is given, but never encountered (for example, if the destination is on a program branch that is never taken), the CodeView debugger executes to the end of the program.

If you enter the Go command and the debugger does not encounter a breakpoint, the entire program is executed and the following message is displayed:

```
Program terminated normally (number)
```

The *number* in parentheses is the value returned by the program (sometimes called the exit or "errorlevel" code).

■ Mouse

To execute the Go command with no destination, point to Go on the menu bar and press either button.

To execute the Goto variation of the Go command, point to the source line or instruction you wish to go to; then press the right button. The highlight marking the current location will move to the source line or instruction you pointed to (unless a breakpoint is encountered first). The CodeView debugger will sound a warning and take no action if you try to go to a comment line or other source line that does not correspond to code.

If the line you wish to go to is in another module, you can use the Load command from the Files menu to load the source file for the other module. Then point to the destination line and press the right button.

■ Keyboard

To use a keyboard command to execute the Go command with no destination, press the F5 key. This works in both window and sequential modes.

To execute the Goto variation of the Go command, move the cursor to the source line or instruction you wish to go to. If the cursor is in the dialog window, first press the F6 key to move the cursor to the display window. When the cursor is at the appropriate line in the display window, press the F7 key. The highlight marking the current location will move to the source line or instruction you pointed to (unless a breakpoint is encountered first). The CodeView debugger will sound a warning and take no action if you try to go to a comment line or other source line that does not correspond to code.

If the line you wish to go to is in another module, you can use the Load command from the Files menu to load the source file for the other module. Then move the cursor to the destination line and press the F7 key.

■ Dialog

To execute the Go command using a dialog command, enter a command line with the following syntax:

G [[*breakaddress*]]

If the command is given with no argument, execution continues until a breakpoint or the end of the program is encountered.

The Goto form of the command can be given by specifying *breakaddress*. The *breakaddress* can be given as a symbol, a line number, or an address in the *segment*:*offset* format. If the offset address is given without a segment, the address in the **CS** register is used as the default segment. If you give *breakaddress* as a line number, but the corresponding source line is a comment, declaration, or blank line, the following message appears:

```
No code at this line number
```

■ Examples

The following examples show the Go command in sequential mode. In window mode, there would be no output from the commands, but the display would be updated to show changes caused by the command.

```
>G

Program terminated normally (0)
>
```

The above example passes control to the instruction pointed to by the current values of the **CS** and **IP** registers. No breakpoint is encountered, so the CodeView debugger executes to the end of the program, where it prints a termination message and the exit code returned by the program (0 in the example).

```
>S+
source
>G BUBBLE
17:             IF (N .LE. 1) GOTO 101
>
```

In the above example, the display mode is first set to source (**S+**). See Chapter 9, "Examining Code," for information on setting the display mode. When the Go command is entered, the CodeView debugger starts program execution at the current address and continues until it reaches the start of the subprogram BUBBLE.

```
>S+
mixed
>G .22
AX=02F4  BX=0002  CX=00A8  DX=0000  SP=3036  BP=3042  SI=0070  DI=40E0
DS=49B7  ES=49B7  SS=49B7  CS=3FB0  IP=0141  NV UP EI PL NZ NA PO NC
22:             X(I) = X(J)
3FB0:0141 8B7608        MOV     SI,Word Ptr [BP+08]          SS:304A=0070
>
```

The above example passes execution control to the program at the current address and executes to the address of source line 22. If the address with the breakpoint is never encountered (for example, if the program has less than 22 lines, or if the breakpoint is on a program branch that is never taken), the CodeView debugger executes to the end of the program.

```
>S-
assembly
>G #2A8
AX=0049  BX=0049  CX=028F  DX=0000  SP=12F2  BP=12F6  SI=04BA  DI=1344
DS=5DAF  ES=5DAF  SS=5DAF  CS=58BB  IP=02A8  NV UP EI PL NZ NA PE NC
58BB:02A8 98            CBW
>
```

The above example executes to address CS:#2A8. Since no segment address is given, the **CS** register is assumed.

5.5 Execute Command

The Execute command is similar to the Go command with no arguments, except that it executes in slow motion (several source lines per second). Execution starts at the current address and continues to the end of the program or until a breakpoint, tracepoint, or watchpoint is reached. You can also stop automatic program execution by pressing any key or a mouse button.

■ Mouse

To execute code in slow motion with the mouse, point to Run on the menu bar, press a mouse button and drag the highlight down to the Execute selection, then release the button.

■ Keyboard

To execute code in slow motion with a keyboard command, press ALT-R to open the Run menu, then press ALT-E to select Execute.

■ Dialog

To execute code in slow motion using a dialog command, enter a command line with the following syntax:

E

You cannot set a destination for the Execute command as you can for the Go command.

In sequential mode, the output from the Execute command depends on the display mode (source, assembly, or mixed). In assembly or mixed mode, the command executes one instruction at a time. The command displays the current status of the registers and the instruction. In mixed mode, it will also show a source line if there is one at the instruction. In source mode, the command executes one source line at a time, displaying the lines as it executes them.

Important

The Execute command has the same command letter (**E**) as the Enter command. If the command has at least one argument, it is interpreted as Enter; if not, it is interpreted as Execute.

5.6 Restart Command

The Restart command restarts the current program. The program is ready to be executed just as if you had restarted the CodeView debugger. Program variables are re-initialized, but any existing breakpoints or watch statements are retained. The pass count for all breakpoints is reset to 1. Any program arguments are also retained, though they can be changed with the dialog version of the command.

The Restart command can only be used to restart the current program. If you wish to load a new program, you must exit and restart the CodeView debugger with the new program name.

■ Mouse

To restart the program with the mouse, point to Run on the menu bar, press a mouse button and drag the highlight down to the Restart or Start selection, then release the button. The program will be restarted. If the Restart selection is chosen, the program will be ready to start executing from the beginning. If the Start selection is chosen, the program starts executing from the beginning and continues until a breakpoint or the end of the program is encountered.

■ Keyboard

To restart the program with a keyboard command, press ALT-R to open the Run menu, then press either ALT-R to select Restart or ALT-S to select Start. The program will be restarted. If the Restart selection is chosen, the program will be ready to start executing from the beginning. If the Start selection is chosen, the program starts executing from the beginning and continues until a breakpoint or the end of the program is encountered.

■ Dialog

To restart the program with a dialog command, enter a command line with the following syntax:

L [[*arguments*]]

When you restart using the dialog version of the command, the program will be ready to start executing from the beginning. If you want to restart with new program arguments, you can give optional *arguments*. You cannot specify new arguments with the mouse or keyboard version of the command.

Note

The command letter **L** is a mnemonic for Load, but the command should not be confused with the Load selection from the File menu, since that selection only loads a source file without restarting the program.

■ Examples

```
>L
>
```

The above example starts the current executable file, retaining any breakpoints, watchpoints, tracepoints, and arguments.

```
>L 6
>
```

The above example restarts the current executable file, but with 6 as the new program argument.

Chapter 6
Examining Data and Expressions

6.1 Introduction 103
6.2 Display Expression Command 103
6.3 Examine Symbols Command 109
6.4 Dump Commands 112
6.4.1 Dump 114
6.4.2 Dump Bytes 114
6.4.3 Dump ASCII 115
6.4.4 Dump Integers 116
6.4.5 Dump Unsigned Integers 117
6.4.6 Dump Words 117
6.4.7 Dump Double Words 118
6.4.8 Dump Short Reals 119
6.4.9 Dump Long Reals 120
6.4.10 Dump 10-Byte Reals 121
6.5 Register Command 122
6.6 8087 Command 123

6.1 Introduction

The CodeView debugger provides several commands for examining different kinds of data, including expressions, variables, memory, and registers. The data-evaluation commands discussed in this chapter are summarized below:

Command	Action
Display Expression (**?**)	Evaluates and displays the value of symbols or expressions
Examine Symbol (**X?**)	Displays the addresses of symbols
Dump (**D**)	Displays sections of memory containing data; there are several variations for examining different kinds of data
Register (**R**)	Shows the current values of each register and each flag
8087 (**7**)	Shows the current value in the 8087 or 80287 register

6.2 Display Expression Command

The Display Expression command displays the value of CodeView expressions.

A CodeView expression can be any valid FORTRAN expression consisting of function calls, intrinsic functions, numbers, symbols, addresses, or operators. The CodeView debugger supports a subset of the FORTRAN operators, as explained in Chapter 4, "Using Dialog Commands." The simplest form of expression is an identifier representing a single variable name, routine or subprogram name, or symbol.

Note

FORTRAN subroutines can be used in expressions, but the value returned by the expression will be meaningless.

In addition to its primary purpose of displaying values, the Display Expression command can also set values as a side effect. For example, if you give the expression `N = N + 1` with the Display Expression command, the value of `N` will be incremented and displayed.

You can specify the format in which the values of expressions are displayed by the Display Expression command. Type a comma after the expression, followed by a CodeView format specifier. The format specifiers used in the CodeView debugger are a subset of those used by the C **printf** function. They are listed in Table 6.1.

Table 6.1
CodeView Format Specifiers

Character	Output Format	Sample Expression	Sample Output
d	Signed decimal integer	`?40000,d`	`40000`
i	Signed decimal integer	`?40000,i`	`40000`
u[1]	Unsigned decimal integer	`?40000,u`	`40000`
o	Unsigned octal integer	`?40000,o`	`116100`
x or **X**[2]	Hexadecimal integer	`?40000,x`	`9c40`
f	Signed value in floating-point decimal format with six decimal places	`?3./2.,f`	`1.500000`
e or **E**[3]	Signed value in scientific-notation format with up to six decimal places (trailing zeros and decimal point are truncated)	`?3./2.,e`	`1.500000e+000`
g or **G**[3]	Signed value with floating-point decimal format (**f**) or scientific-notation format (**g** or **G**), whichever is more compact	`?3./2.,g`	`1.5`
c	Single character	`?65,c`	`A`
s	Characters printed up to the first null character	`?'String',s`	`'String'`

[1] FORTRAN has no unsigned data types. Using an unsigned format specifier has no effect on the output of positive numbers, but causes negative numbers to be output as positive values.

[2] Hexadecimal letters are uppercase if the type is **X** and lowercase if the type is **x**.

[3] The "E" in scientific-notation numbers is uppercase if the type is **E** or **G**, lowercase if the type is **e** or **g**.

If no format specifier is given, single- and double-precision real numbers are displayed as if the format specifier had been given as **g**. (If you are familiar with the C language, you should note that the **n** and **p** format specifiers and the **F** and **H** prefixes are not supported by the CodeView debugger even though they are supported by the C **printf** function.)

The prefix **h** can be used with the integer format specifiers (**d**, **o**, **u**, **x**, and **X**) to specify an **INTEGER*2**. The prefix **l** can be used with the same types to specify an **INTEGER*4**. For example, the command `?100000,ld` produces the output `100000`. However, the command `?100000,hd` evaluates only the **INTEGER*2** part of the value (the low-order 2 bytes), producing the output `-31072`.

The Display Expression command does not work for programs assembled with Microsoft Macro Assembler versions 4.0 and earlier, because the assembler does not write information to the object file about the type size of each variable. Use the Dump command instead.

When calling a FORTRAN subroutine that uses alternate returns, the value of the return labels in the actual parameter list must be 0. For example, the subroutine call

```
CALL PROCESS (I,*10,J,*20,*30)
```

must be called from the debugger as

```
?PROCESS(IARG1,0,IARG2,0,0)
```

Using other values as return labels will cause the error

```
Type clash in function argument
```

or

```
Unknown symbol
```

■ Mouse

To display the value of an expression with the mouse, point to View on the menu bar, press a mouse button and drag the highlight down to the Evaluate selection, then release the button. A dialog box appears, asking for the expression you want to evaluate. Type the expression and press the ENTER key (or a mouse button). You can add a comma and a format specifier if you want to specify the output format. The value of the expression will appear in the dialog window.

■ Keyboard

To display the value of an expression using a keyboard command, press ALT-V to open the View menu, then press ALT-E to select Evaluate. A dialog box appears, asking for the expression you want to evaluate. Type the expression and press the ENTER key. You can add a comma and a format specifier if you want to specify the output format. The value of the expression will appear in the dialog window.

■ Dialog

To display the value of an expression using a dialog command, enter a command line with the following syntax:

? *expression*[[,*format*]]

The *expression* is any valid CodeView expression, and the optional *format* is a CodeView format specifier.

■ Examples

The examples assume that the source file contains the following variable declarations:

```
INTEGER*2 MILES
REAL HOURS
CHARACTER*16 STR
MILES = 500
HOURS = 5.32
STR = 'Here is a string'
```

Assume also that the program has been executed to the point where all these variables have been assigned values.

```
>? MILES
500
>? MILES,x
01f4
>? MILES,o
764
>
```

The above example displays the value of the variable MILES, first in the default decimal format, then in hexadecimal, and then in octal.

```
>? 92,X
005C
>? 109*37,o
7701
>? 'T'
84
>? 118,c
v
>
```

The above example illustrates how the CodeView debugger can be used as a calculator. You can convert between radixes, calculate the value of constant expressions, or check ASCII equivalents.

```
>X?STRING
44CD:07EA
>DA #07EA
44CD:07EA  Here is a string...
>? STRING,s
'Here is a string'
>
```

The above example shows how to examine strings. One method is to evaluate the variable that points to the string, then dump the values at that address (the Dump command is explained in Section 6.4). A more direct method is to use the `s` format specifier.

```
>? MILES
500
>? HOURS
5.320000
>? MILES/HOURS
93.984962
>? MILES/HOURS,e
9.398496e+001
>
```

The above example displays the value of the symbols `MILES` and `HOURS`. The two variables are then combined to calculate miles per hour. The real number is shown both in real-number format and in scientific notation using the `e` format specifier.

```
>? MILES
500
>? MILES+1
501
>? MILES=600
600
>
```

The above example shows how the Display Expression command can be used to change the values of variables.

6.3 Examine Symbols Command

The Examine Symbols command displays the names and addresses of symbols, and the names of modules, defined within a program. You can specify the symbol or group of symbols you want to examine by module, procedure, or symbol name.

■ Mouse

This command cannot be executed with the mouse.

■ Keyboard

This command cannot be executed with a keyboard command.

■ Dialog

To view the addresses of symbols using a dialog command, enter a command line in one of the following formats:

X*
X? [[*module*!]] [[*routine*.]] [[*symbol*]] [[*]]

The syntax combinations are listed in more detail below:

Syntax	Display
X?*module*!*routine*.*symbol*	The specified *symbol* in the specified *routine* in the specified *module*
X?*module*!*routine*.*	All symbols in the specified *routine* in the specified *module*
X?*module*!*symbol*	The specified *symbol* in the specified *module*; symbols within routines are not found
X?*module*!*	All symbols in the specified *module*
X?*routine*.*symbol*	The specified *symbol* in the specified *routine*; looks for *routine* first in the current module, then in other modules from first to last
X?*routine*.*	All symbols in the specified *routine*; looks for *routine* first in the current module, then in other modules from first to last
X?*symbol*	Looks for the specified *symbol* in this order: 1. In the current routine 2. In the current module 3. In other modules, from first to last
X?*	All symbols in the current routine
X*	All module names

To view global symbols when debugging assembly language modules, you must specify the module. For example, use `X?module` instead of `X?*`.

■ Examples

In the following examples, assume that the program being examined is called FRUST.EXE, and that it consists of four modules: FRUST.FOR, FRUST1.FOR, FRUST2.FOR, and FRUST3.FOR. Assume that the current routine is main within the FRUST.FOR module.

```
>X*
FRUST.OBJ
FRUST1.OBJ
FRUST2.OBJ
FRUST3.OBJ
c:\lib\LLIBFORE.LIB(fixups)
c:\lib\LLIBFORE.LIB(crt0)
c:\lib\LLIBFORE.LIB(chkstk)
c:\lib\LLIBFORE.LIB(wr)
.
.
.
c:\lib\LLIBFORE.LIB(txtmode)
c:\lib\LLIBFORE.LIB(_creat)
```

The above example lists the three modules called by the program. The library files called by the program are also listed.

```
>X?T
        520D:0DE4 REAL*4          T
```

The above example shows the address of the variable T in the current module.

```
>X?FRUST3!MULTPI.*
4B28:0005 INTEGER*4         MULTPI()
                [BP+000A]                       V
                [BP+0006]                       X
                [BP-0004] INTEGER*4             MULTPI
```

The above example lists the symbols in the function MULTPI, located in module FRUST3. Variables that are local to the function are indented under the function. You wouldn't need to specify the module if FRUST3 were the current module.

```
>X?FRUST2!SAREA.*
4B15:000E void                  SAREA()
                    [BP+0012]                       R1
                    [BP+000E]                       R2
                    [BP+000A]                       H
                    [BP+0006]                       T
                    520D:0DEC REAL*4                S12
                    520D:0DE8 REAL*4                U
```

The above example shows all the symbols in the routine SAREA in the module FRUST2. Because SAREA is a subroutine instead of a function, the word void appears where function return value types are shown.

6.4 Dump Commands

The CodeView debugger has several commands for dumping data from memory to the screen (or other output device). The dump commands are listed below:

Command	Command Name
D	Dump (size is the default type)
DB	Dump Bytes
DA	Dump ASCII
DI	Dump Integers
DU	Dump Unsigned Integers
DW	Dump Words
DD	Dump Double Words
DS	Dump Short Reals
DL	Dump Long Reals
DT	Dump 10-Byte Reals

■ Mouse

The Dump commands cannot be executed with the mouse.

■ Keyboard

The Dump commands cannot be executed with keyboard commands.

■ Dialog

To execute any Dump command using a dialog command, enter a command line with the following syntax:

D[[*type*]] [[*address* | *range*]]

The *type* is a one-letter specifier that indicates the type of the data to be dumped. The dump commands expect either a starting *address* or a *range* of memory. If the starting *address* is given, the commands assume a default range (usually 128 bytes) starting at *address*. If *range* is given, the commands dump from the start to the end of *range*.

If neither *address* nor *range* is given, the commands assume the current dump address as the start of the range and the default size associated with the size of the object as the length of the range. Most Dump commands have a default range size of 128 bytes, but the Dump Real commands have a default range size of one real number.

The current dump address is the byte following the last byte specified in the previous Dump command. If no Dump command has been used during the session, the dump address is the start of the data segment (**DS**). For example, if you enter the Dump Words command with no argument as the first command of a session, the CodeView debugger displays the first 64 words (128 bytes) of data declared in the data segment. If you repeat the same command, the debugger displays the next 64 words following the ones dumped by the first command.

Note

If the value in memory cannot be evaluated as a real number, the Dump commands that display real numbers (Dump Short Reals, Dump Long Reals, or Dump 10-Byte Reals) will display a number containing one of the following character sequences: #NAN, #INF, or #IND. NAN (not a number) indicates that the data cannot be evaluated as a real number. INF (infinity) indicates that the data evaluates to infinity. IND (indefinite) indicates that the data evaluates to an indefinite number.

Sections 6.4.1 – 6.4.10 discuss the variations of the Dump commands in order of the size of data they display.

6.4.1 Dump

■ Syntax

D [[*address* | *range*]]

The Dump command displays the contents of memory at the specified *address* or in the specified *range* of addresses. The command dumps data in the format of the default type. The default type is the last type specified with a Dump, Enter, Watch Memory, or Tracepoint Memory command. If none of these commands has been entered during the session, the default type is bytes.

The Dump command displays one or more lines, depending on the address or range specified. Each line displays the address of the first item displayed. The Dump command must be separated by at least one space from any *address* or *range* value. For example, to dump memory starting at symbol `a`, use the command `D a`, not `Da`. The second syntax would be interpreted as the Dump ASCII command.

6.4.2 Dump Bytes

■ Syntax

DB [[*address* | *range*]]

The Dump Bytes command displays the hexadecimal and ASCII values of the bytes at the specified *address* or in the specified *range* of addresses. The command displays one or more lines, depending on the address or range supplied.

Each line displays the address of the first byte in the line, followed by up to 16 hexadecimal byte values. The byte values are immediately followed by the corresponding ASCII values. The hexadecimal values are separated by spaces, except the eighth and ninth values, which are separated by a dash (–). ASCII values are printed without separation. Unprintable ASCII values (less than 32 or greater than 126) are displayed as dots. No more than 16 hexadecimal values are displayed in a line. The command displays values and characters until the end of the *range* or, if no *range* is given, until the first 128 bytes have been displayed.

■ Example

```
>DB 0 36
3D5E:0000 53 6F 6D 65 20 6C 65 74-74 65 72 73 20 61 6E 64 Some letters and
3D5E:0010 20 6E 75 6D 62 65 72 73-3A 00 10 EA 89 FC FF EF  numbers:.......
3D5E:0020 00 F0 00 CA E4                                   .....
>
```

The above example displays the byte values from DS:0 to DS:36 (DS:#24). The data segment is assumed if no segment is given. ASCII characters are shown on the right.

6.4.3 Dump ASCII

■ Syntax

DA [[*address* | *range*]]

The Dump ASCII command displays the ASCII characters at a specified *address* or in a specified *range* of addresses. The command displays one or more lines of characters, depending on the *address* or *range* specified.

If no ending address is specified, the command dumps either 128 bytes or all bytes preceding the first null byte, whichever comes first. Up to 64 characters per line are displayed. Unprintable characters, such as carriage returns and line feeds, are displayed as dots. ASCII characters less than 32 and greater than 126 are unprintable.

■ Examples

```
>DA 0
3D7C:0000  Some letters and numbers:
>
```

The above example displays the ASCII values of the bytes starting at DS:0. Since no ending address is given, values are displayed up to the first null byte.

```
>DA 0 36
3D7C:0000  Some letters and numbers:.....
>
```

In the above example, an ending address is given, so the characters from DS:0 to DS:36 (DS:#24) are shown. Unprintable characters are shown as dots.

6.4.4 Dump Integers

■ Syntax

DI [[*address* | *range*]]

The Dump Integers command displays the signed decimal values of the words (2-byte values) starting at *address* or in the specified *range* of addresses. The command displays one or more lines, depending on the address or range specified. Each line displays the address of the first integer in the line, followed by up to eight signed decimal words. The values are separated by spaces. The command displays values until the end of the *range* or until the first 64 2-byte integers have been displayed, whichever comes first.

Note

In this manual an integer is considered a 2-byte value, since that is the integer size the CodeView debugger assumes. Note that a default FORTRAN integer is a 4-byte value.

■ Example

```
>DI 0 36
3D5E:0000   28499  25965  27680  29797  25972  29554  24864  25710
3D5E:0010   28192  28021  25954  29554     58  -5616   -887  -4097
3D5E:0020   -4096 -13824   2532
>
```

The example shown above displays the byte values from DS:0 to DS:36 (DS:#24). Compare the signed decimal numbers at the end of this dump with the same values shown as unsigned integers in Section 6.4.5.

6.4.5 Dump Unsigned Integers

■ Syntax

DU [[*address* | *range*]]

The Dump Unsigned Integers command displays the unsigned decimal values of the words (2-byte values) starting at *address* or in the specified *range* of addresses. The command displays one or more lines, depending on the address or range specified. Each line displays the address of the first unsigned integer in the line, followed by up to eight decimal words. The values are separated by spaces. The command displays values until the end of the *range* or until the first 64 unsigned integers have been displayed, whichever comes first.

■ Example

```
>DU 0 36
3D5E:0000   28499  25965  27680  29797  25972  29554  24864  25710
3D5E:0010   28192  28021  25954  29554     58  59920  64649  61439
3D5E:0020   61440  51712   2532
>
```

The example shown above displays the byte values from DS:0 to DS:36 (DS:#24). Compare the unsigned decimal numbers at the end of this dump with the same values shown as signed integers in Section 6.4.4.

6.4.6 Dump Words

■ Syntax

DW [[*address* | *range*]]

The Dump Words command displays the hexadecimal values of the words (2-byte values) starting at *address* or in the specified *range* of addresses. The command displays one or more lines, depending on the address or range specified. Each line displays the address of the first word in the line, followed by up to eight hexadecimal words. The hexadecimal values are separated by spaces. The command displays values until the end of the *range* or until the first 64 words have been displayed, whichever comes first.

■ **Example**

```
>DW 0 36
3D5E:0000  6F53 656D 6C20 7465 6574 7372 6120 646E
3D5E:0010  6E20 6D75 6562 7372 003A EA10 FC89 EFFF
3D5E:0020  F000 CA00 09E4
>
```

The above example displays the word values from DS:0 to DS:36 (DS:#24). No more than eight values per line are displayed.

6.4.7 Dump Double Words

■ **Syntax**

DD [[*address* | *range*]]

The Dump Double Words command displays the hexadecimal values of the double words (4-byte values) starting at *address* or in the specified *range* of addresses.

The command displays one or more lines, depending on the address or range specified. Each line displays the address of the first double word in the line, followed by up to four hexadecimal double-word values. The words of each double word are separated by a colon. The values are separated by spaces. The command displays values until the end of the *range* or until the first 32 double words have been displayed, whichever comes first.

■ **Example**

```
>DD 0 36
3D5E:0000  656D:6F53 7465:6C20 7372:6574 646E:6120
3D5E:0010  6D75:6E20 7372:6562 EA10:003A EFFF:FC89
3D5E:0020  CA00:F000 6F73:09E4
>
```

The above example displays the double-word values from DS:0 to DS:36 (DS:#24). No more than four double-word values per line are displayed.

6.4.8 Dump Short Reals

■ Syntax

DS [[*address* | *range*]]

The Dump Short Reals command displays the hexadecimal and decimal values of the short (4-byte) floating-point numbers at *address* or in the specified *range* of addresses.

The command displays one or more lines, depending on the address or range specified. Each line displays the address of the floating-point number in the first column. Next, the hexadecimal values of the bytes in the number are shown, followed by the decimal value of the number. The hexadecimal values are separated by spaces.

The decimal value has the following form:

[[−]]*digit.decimaldigits***E**{**+** | **−**}*exponent*

If the number is negative, it will have a minus sign; positive numbers have no sign. The first digit of the number is followed by a decimal point. Six decimal places are shown following the decimal point. The letter **E** follows the decimal digits, and marks the start of a three-digit signed *exponent*.

The command displays at least one value. If a *range* is specified, all values in the range are displayed.

■ Example

```
>DS SPI
5E68:0100  DB 0F 49 40  3.141593E+000
>
```

The above example displays the short-real floating-point number at the address of the variable SPI. Only one value is displayed per line.

6.4.9 Dump Long Reals

■ Syntax

DL [[*address* | *range*]]

The Dump Long Reals command displays the hexadecimal and decimal values of the long (8-byte) floating-point numbers at the specified *address* or in the specified *range* of addresses.

The command displays one or more lines, depending on the address or range specified. Each line displays the address of the floating-point number in the first column. Next, the hexadecimal values of the bytes in the number are shown, followed by the decimal value of the number. The hexadecimal values are separated by spaces.

The decimal value has the following form:

[[−]]*digit.decimaldigits***E**{**+** | **−**}*exponent*

If the number is negative, it will have a minus sign; positive numbers have no sign. The first digit of the number is followed by a decimal point. Six decimal places are shown following the decimal point. The letter **E** follows the decimal digits, and marks the start of a three-digit signed *exponent*.

The command displays at least one value. If a *range* is specified, all values in the range are displayed.

■ Example

```
>DL LPI
5E68:0200  11 2D 44 54 FB 21 09 40  3.141593E+000
>
```

The above example displays the long-real floating-point number at the address of the variable LPI. Only one value per line is displayed.

6.4.10 Dump 10-Byte Reals

■ Syntax

DT [[*address* | *range*]]

The Dump 10-Byte Reals command displays the hexadecimal and decimal values of the 10-byte floating-point numbers at the specified *address* or in the specified *range* of addresses.

The command displays one or more lines, depending on the address or range specified. Each line displays the address of the floating-point number in the first column. Next, the hexadecimal values of the bytes in the number are shown, followed by the decimal value of the number. The hexadecimal values are separated by spaces.

The decimal value has the following form:

[[−]]*digit.decimaldigits***E**{**+** | **−**}*exponent*

If the number is negative, it will have a minus sign; positive numbers have no sign. The first digit of the number is followed by a decimal point. Six decimal places are shown following the decimal point. The letter **E** follows the decimal digits, and marks the start of a three-digit signed *exponent*.

The command displays at least one value. If a *range* is specified, all values in the range are displayed.

■ Example

```
>DT TPI
5E68:0300  DE 87 68 21 A2 DA 0F C9 00 40  3.141593E+000
>
```

The above example displays the 10-byte-real floating-point number at the address of the variable TPI. Only one number per line is displayed.

6.5 Register Command

The Register command has two functions. It displays the contents of the central processing unit (CPU) registers. It can also change the values of the registers. The display features of the Register command are explained here. The modification features of the command are explained in Chapter 10, "Modifying Code or Data."

■ Mouse

To display the registers with the mouse, point to Options on the menu bar, press a mouse button and drag the highlight down to the Registers selection, then release the button. The register window will appear on the right side of the screen. If the register window is already on the screen, the same command removes it.

■ Keyboard

To display the registers using a keyboard command in window mode, press the F2 key. The register window will appear on the right side of the screen. If the register window is already on the screen, the same command will remove it.

In sequential mode, the F2 key will display the current status of the registers. (This produces the same effect as entering the Register dialog command with no argument.)

■ Dialog

To display the registers in the dialog window (or sequentially in sequential mode), enter a command line with the following syntax:

R

The current values of all registers and flags are displayed (in the dialog window in window mode). The instruction at the address pointed to by the current **CS** and **IP** register values is also shown. (The Register command can also be given with arguments, but only when used to modify registers, as explained in Chapter 10, "Modifying Code or Data.")

If the display mode is source (**S+**) or mixed (**S&**) (see Chapter 9, "Examining Code," for more information), the current source line is also displayed by the Register command. If an operand of the instruction contains memory expressions or immediate data, the CodeView debugger will evaluate operands and show the value to the right of the instruction. If the **CS** and **IP** registers are currently at a breakpoint location, the register display will indicate the breakpoint number.

In sequential mode, the Trace (**T**), Program Step (**P**), and Go (**G**) commands show registers in the same format as the Register command.

■ Examples

```
>S&
mixed
>R
AX=0005  BX=299E  CX=0000  DX=0000  SP=3800  BP=380E  SI=0070  DI=40D1
DS=5067  ES=5067  SS=5067  CS=4684  IP=014F  NV UP EI PL NZ NA PO NC
35:                        VARIAN = (N*SUMXSQ-SUMX**2)/(N-1)
4684:014F 8B5E06        MOV     BX,Word Ptr [BP+06]     ;BR1  SS:3814=299E
>
```

The above example displays all register and flag values, as well as the instruction at the address pointed to by the **CS** and **IP** registers. Since the mode has been set to mixed (**S&**), the current source line is also shown.

```
>S-
assembly
>R
AX=0005  BX=299E  CX=0000  DX=0000  SP=3800  BP=380E  SI=0070  DI=40D1
DS=5067  ES=5067  SS=5067  CS=4684  IP=014F  NV UP EI PL NZ NA PO NC
4684:014F 8B5E06        MOV     BX,Word Ptr [BP+06]     ;BR1  SS:3814=299E
>
```

In the above example, the display mode is set to assembly (**S−**), so no source line is shown. Note the breakpoint number at the right of the last line, indicating that the current address is at Breakpoint 1.

6.6 8087 Command

The 8087 command dumps the contents of the 8087 registers. This command is useful only if you have an 8087 or 80287 coprocessor chip on your system.

Note

This section does not attempt to explain how the registers of the Intel® 8087 and 80287 processors are organized or how they work. In order to interpret the command output, you must learn about the chip from an Intel reference manual or other book on the subject.

■ Mouse

The 8087 command cannot be executed with the mouse.

■ Keyboard

The 8087 command cannot be executed with a keyboard command.

■ Dialog

To display the status of the 8087 or 80287 chip using a dialog command, enter a command line with the following syntax:

7

The current status of the chip is output when you enter the command. In window mode, the output is to the dialog window. If you do not have an 8087 or 80287 chip, all registers in the output will contain 0.

■ Example

```
>7
Control 037F  (Projective closure, Round nearest, 64-bit precision)
                    iem=0 pm=1 um=1 om=1 zm=1 dm=1 im=1
Status  6004  cond=1000 top=4 pe=0 ue=0 oe=0 ze=1 de=0 ie=0
Tag     A1FF  instruction=59380  operand=59360  opcode=D9EE
Stack         Exp  Mantissa           Value
ST(3) special 7FFF 8000000000000000 = + Infinity
ST(2) special 7FFF 0101010101010101 = + Not a Number
ST(1) valid   4000 C90FDAA22168C235 = +3.14159226511039OE+000
ST(0) zero    0000 0000000000000000 = +0.000000000000000E+000
>
```

In the example above, the first line of the dump shows the current closure method, rounding method, and precision. The number 037F is the hexadecimal value in the control register. The rest of the line interprets the bits of the number. The closure method can be either projective (as in the above example) or affine. The rounding method can be either rounding to the nearest even number (as in the example), rounding down, rounding up, or using the chop method of rounding (truncating toward zero). The precision may be 64 bits (as in the example), 53 bits, or 24 bits.

The second line of the display indicates whether each exception mask bit is set or cleared. The masks are interrupt-enable mask (iem), precision mask (pm), underflow mask (um), overflow mask (om), zero-divide mask (zm), denormalized-operand mask (dm), and invalid-operation mask (im).

The third line of the display shows the hexadecimal value of the status register (6004 in the example), then interprets the bits of the register. The condition code (cond) in the example is the binary number 1000. The top of the stack (top) is Register 4 (shown in decimal). The other bits shown are precision exception (pe), underflow exception (ue), overflow exception (oe), zero-divide exception (ze), denormalized-operand exception (de), and invalid-operation exception (ie).

The fourth line of the display first shows the hexadecimal value of the tag register (A1FF in the example). It then gives the hexadecimal values of the instruction (59380), the operand (59360), and the operation code, or opcode, (D9EE).

The fifth line is a heading for the subsequent lines, which contain the contents of each 8087 or 80287 stack register. The registers in the example contain four types of numbers that may be held in these registers. Starting from the bottom, Register 0 contains zero. Register 1 contains a valid real number. Its exponent (in hexadecimal) is 4000 and its mantissa is C90FDAA22168C235. The number is shown in scientific notation in the rightmost column. Register 2 contains a value that cannot be interpreted as a number, and Register 3 contains infinity.

Chapter 7
Managing Breakpoints

7.1 Introduction 129
7.2 Breakpoint Set Command 129
7.3 Breakpoint Clear Command 132
7.4 Breakpoint Disable Command 134
7.5 Breakpoint Enable Command 135
7.6 Breakpoint List Command 136

7.1 Introduction

The CodeView debugger enables you to control program execution by setting breakpoints. A breakpoint is an address that stops program execution each time the address is encountered. By setting breakpoints at key addresses in your program, you can "freeze" program execution and examine the status of memory or expressions at that point.

The commands listed below control breakpoints:

Command	Action
Breakpoint Set (**BP**)	Sets a breakpoint and, optionally, a pass count and break commands
Breakpoint Clear (**BC**)	Clears one or more breakpoints
Breakpoint Disable (**BD**)	Disables one or more breakpoints
Breakpoint Enable (**BE**)	Enables one or more breakpoints
Breakpoint List (**BL**)	Lists all breakpoints

In addition to these commands, the Watchpoint (**WP**) and Tracepoint (**TP**) commands can be used to set conditional breakpoints. These commands are explained in Chapter 8, "Managing Watch Statements." The Breakpoint commands are discussed in Sections 7.2–7.6.

7.2 Breakpoint Set Command

The Breakpoint Set command creates a breakpoint at a specified address. Any time a breakpoint is encountered during program execution, the program halts and waits for a new command.

The CodeView debugger allows up to 20 breakpoints (0 through 19). Each new breakpoint is assigned the next available number. Breakpoints remain in memory until you delete them (see Section 7.3 for more information) or until you quit the debugger. They are not canceled when you restart the program. This enables you to set up a complicated series of breakpoints, then execute through the program several times without resetting the breakpoints.

If you try to set a breakpoint at a comment line or other source line that does not correspond to code, the CodeView debugger displays the following message:

```
No code at this line number
```

■ Mouse

To set a breakpoint with the mouse, point to the source line or instruction where you want to set the breakpoint, then click the left button. The line will be displayed in high-intensity text, and will remain so until you remove or disable the breakpoint.

■ Keyboard

To set a breakpoint with a keyboard command in window mode, move the cursor to the source line or instruction where you want to set a breakpoint. You may have to press the F6 key to move the cursor to the display window. When the cursor is on the appropriate source line, press the F9 key. The line will be displayed in high-intensity text, and will remain so until you remove the breakpoint.

In sequential mode, the F9 key can be used to set a breakpoint at the current location. You must use the dialog version of the command to set a breakpoint at any other location.

■ Dialog

To set a breakpoint using a dialog command, enter a command line with the following syntax:

BP [[*address* [[*passcount*]] [["*commands*"]]]]

If no *address* is given, a breakpoint is created on the current source line in source mode, or on the current instruction in assembly mode. You can specify the *address* in the *segment*:*offset* format or as a source line, a routine name, or a label. If you give an offset address, the code segment is assumed.

The dialog version of the command is more powerful than the mouse or keyboard version in that it allows you to give a *passcount* and a string of *commands*. The *passcount* specifies the first time the breakpoint is to be taken. For example, if the pass count is 5, the breakpoint will be ignored the first four times it is encountered, and taken the fifth time.

The *commands* are a list of dialog commands enclosed in quotation marks (" ") and separated by semicolons (;). For example, if you specify the commands as "? code;T", the CodeView debugger will automatically display the value of the variable code and then execute the Trace command each time the breakpoint is encountered. The Trace and Display Expression commands are described in Chapter 6, "Examining Data and Expressions," and Chapter 5, "Executing Code," respectively.

In window mode, a breakpoint entered with a dialog command has exactly the same effect as one created with a window command. The source line or instruction corresponding to the breakpoint location is shown in high-intensity text.

In sequential mode, information about the current instruction will be displayed each time you execute to a breakpoint. The register values, the current instruction, and the source line may be shown, depending on the display mode. See Chapter 9, "Examining Code," for more information about display modes.

When a breakpoint address is shown in the assembly-language format, the breakpoint number will be shown as a comment to the right of the instruction. This comment appears even if the breakpoint is disabled (but not if it is deleted).

■ Examples

```
>BP .19 10
>
```

The above example creates a breakpoint at Line 19 of the current source file (or if there is no executable statement at Line 19, at the first executable statement after Line 19). The breakpoint is passed over 9 times before being taken on the 10th pass.

```
>BP STATS 10 "?COUNTER = COUNTER + 1;G"
>
```

The above example creates a breakpoint at the address of the routine STATS. The breakpoint is passed over 9 times before being taken on the 10th pass. Each time execution stops for the breakpoint, the quoted commands are executed. The Display Expression command increments COUNTER, then the Go command restarts execution. If COUNTER is set to 0 when the breakpoint is set, this has the effect of counting the number of times the breakpoint address is passed.

```
>S-
assembly
>BP #0a94
>G
AX=0006  BX=304A  CX=000B  DX=465D  SP=3050  BP=3050  SI=00BB  DI=40D1
DS=5064  ES=5064  SS=5064  CS=46A2  IP=0A94  NV UP EI PL NZ NA PE NC
46A2:0A94 7205          JB      __chkstk+13 (0A9B)      ;BR1
>
```

The above example first sets the mode to assembly, then creates a breakpoint at the offset address #0A94 in the default (**CS**) segment. The Go command (**G**) is then used to execute to the breakpoint. Note that in the output to the Go command, the breakpoint number is shown as an assembly-language comment (;BR1) to the right of the current instruction. The Go command displays this output only in sequential mode; in window mode no assembly-language information appears.

7.3 Breakpoint Clear Command

The Breakpoint Clear command permanently removes one or more previously set breakpoints.

■ Mouse

To clear a single breakpoint with the mouse, point to the breakpoint line or instruction you want to clear. Breakpoint lines are shown in high-intensity text. Press the left mouse button. The line will be shown in normal text to indicate that the breakpoint has been removed.

To remove all breakpoints with the mouse, point to Run on the menu bar, press a mouse button and drag the highlight down to the Clear Breakpoints selection, then release the button.

■ Keyboard

To clear a single breakpoint with a keyboard command, move the cursor to the breakpoint line or instruction you want to clear. Breakpoint lines are shown in high-intensity text. Press the F9 key. The line will be shown in normal text to indicate that the breakpoint has been removed.

To remove all breakpoints using a keyboard command, press ALT-R to open the Run menu, then press ALT-C to select Clear Breakpoints.

■ Dialog

To clear breakpoints using a dialog command, enter a command line with the following syntax:

BC *list*
BC *

If *list* is specified, the command removes the breakpoints named in the list. The *list* can be any combination of integer values from 0 to 19. You can use the Breakpoint List command (**BL**) if you need to see the numbers for each existing breakpoint. If an asterisk (*) is given as the argument, all breakpoints are removed.

■ Examples

```
>BC  0  4  8
>
```

The above example removes breakpoints 0, 4, and 8.

```
>BC  *
>
```

The above example removes all breakpoints.

7.4 Breakpoint Disable Command

The Breakpoint Disable command temporarily disables one or more existing breakpoints. The breakpoints are not deleted. They can be restored at any time using the Breakpoint Enable command (**BE**).

When a breakpoint is disabled in window mode, it is shown in the display window with normal text. When a breakpoint is enabled, it is shown in high-intensity text.

Note

All disabled breakpoints are automatically enabled whenever you restart the program being debugged. The program can be restarted with the Start or Restart selection from the Run menu, or with the Restart dialog command (**L**). See Chapter 5, "Executing Code."

■ Mouse

The Breakpoint Disable command cannot be executed with the mouse.

■ Keyboard

The Breakpoint Disable command cannot be executed with a keyboard command.

■ Dialog

To disable breakpoints using a dialog command, enter a command line with the following syntax:

BD *list*
BD *

If *list* is specified, the command disables the breakpoints named in the list. The *list* can be any combination of integer values from 0 to 19. Use the Breakpoint List command (**BL**) if you need to see the numbers for each

existing breakpoint. If an asterisk (*) is given as the argument, all breakpoints are disabled.

The window commands for setting and clearing breakpoints can also be used to enable or clear disabled breakpoints.

■ Examples

```
>BD 0 4 8
>
```

The above example disables breakpoints 0, 4, and 8.

```
>BD *
>
```

The above example disables all breakpoints.

7.5 Breakpoint Enable Command

The Breakpoint Enable command enables breakpoints that have been temporarily disabled with the Breakpoint Disable command.

■ Mouse

To enable a disabled breakpoint with the mouse, point to the source line or instruction of the breakpoint, then click the left button. The line will be displayed in high-intensity text, and will remain so until you remove or disable the breakpoint. This is the same as creating a new breakpoint at that location.

■ Keyboard

To enable a disabled breakpoint using a keyboard command, move the cursor to the source line or instruction of the breakpoint, then press the F9 key. The line will be displayed in high-intensity text, and will remain so until you remove or disable the breakpoint. This is the same as creating a new breakpoint at that location.

■ Dialog

To enable breakpoints using a dialog command, enter a command line with the following syntax:

BE *list*
BE *

If *list* is specified, the command enables the breakpoints named in the list. The *list* can be any combination of integer values from 0 to 19. Use the Breakpoint List command (**BL**) if you need to see the numbers for each existing breakpoint. If an asterisk (*) is given as the argument, all breakpoints are enabled. The CodeView debugger ignores all or part of the command if you try to enable a breakpoint that is not disabled.

■ Examples

```
>BE 0 4 8
>
```

The above example enables breakpoints 0, 4, and 8.

```
>BE*
>
```

The above example enables all disabled breakpoints.

7.6 Breakpoint List Command

The Breakpoint List command lists current information about all breakpoints.

■ Mouse

The Breakpoint List command cannot be executed with the mouse.

■ Keyboard

The Breakpoint List command cannot be executed with a keyboard command.

■ Dialog

To list breakpoints using a dialog command, enter a command line with the following syntax:

BL

The command displays the breakpoint number, the enabled status, the address, the routine, and the line number. If the breakpoint does not fall on a line number, an offset is shown from the nearest previous line number. The pass count and break commands are shown if they have been set. The status can be e for enabled, d for disabled. If no breakpoints are currently defined, nothing is displayed.

■ Example

```
>BL
0  e  56C4:0105   _ARCTAN:10
1  d  56C4:011E   _ARCTAN:19                (pass = 10) "T;T"
2  e  56C4:00FD   _ARCTAN:9+6
>
```

In the above example, Breakpoint 0 is enabled at address 56C4:0105. This address is in routine ARCTAN and is at line 10 of the current source file. No pass count or break commands have been set.

Breakpoint 1 is currently disabled, as indicated by the d after the breakpoint number. It also has a pass count of 10, meaning that the breakpoint will not be taken until the 10th time it is encountered. The command string at the end of the line indicates that each time the breakpoint is taken, the Trace command will automatically be executed twice.

The line number for Breakpoint 2 has an offset. The address is #6 bytes beyond the address for line number 9 in the current source file. This means that the breakpoint was probably set in assembly mode, since it is difficult to set a breakpoint anywhere except on a source line in source mode.

Chapter 8
Managing Watch Statements

8.1 Introduction 141

8.2 Setting Watch-Expression and Watch-Memory Statements 142

8.3 Setting Watchpoints 146

8.4 Setting Tracepoints 150

8.5 Deleting Watch Statements 155

8.6 Listing Watchpoints and Tracepoints 156

8.1 Introduction

Watch Statement commands are among the Microsoft CodeView debugger's most powerful features. They enable you to set, delete, and list watch statements. Watch statements are specifications that describe expressions or areas of memory to watch. Some watch statements also specify conditional breakpoints that may or may not be taken, depending on the value of the expression or memory area.

The Watch Statement commands are summarized below:

Command	Action
Watch (**W**)	Sets an expression or range of memory to be watched
Watchpoint (**WP**)	Sets a conditional breakpoint that will be taken when the expression becomes nonzero (true)
Tracepoint (**TP**)	Sets a conditional breakpoint that will be taken when a given expression or range of memory changes
Watch Delete (**Y**)	Deletes one or more watch statements
Watch List (**W**)	Lists current watch statements

Watch statements are like breakpoints in that they remain in memory until you specifically remove them or quit the CodeView debugger. They are not canceled when you restart the program being debugged. This lets you set a complicated series of watch statements, then execute through the program several times without resetting the watch statements.

In window mode, Watch Statement commands can be entered either in the dialog window or with menu selections. Current watch statements are shown in a watch window that appears between the menu bar and the source window.

In sequential mode, the Watch, Tracepoint, and Watchpoint commands can be used, but since there is no watch window, you cannot see the watch statements and their values. You must use the Watch List command to examine the current watch statements.

Note

In order to set a watch statement containing a local variable, you must be in the routine where the variable is defined. If the current line is not in the routine, the CodeView debugger displays the message `Unknown symbol`. When you exit from a routine containing a local variable referenced in a watch statement, the value of the statement is displayed as `Unknown symbol`. When you reenter the routine, the local variable will again have a value. You can avoid this limitation by using the period operator to specify both the routine and the variable. For example, enter `CIRCLE.RADIUS` instead of just `RADIUS`.

Because optimization can affect variable values, especially variables in loops, be sure to turn off optimization with the **/Od** option when compiling programs for use with the CodeView debugger. Otherwise, the variable values displayed by Watch and Trace statements may be incorrect.

8.2 Setting Watch-Expression and Watch-Memory Statements

The Watch command is used to set a watch statement that describes an expression or a range of addresses in memory. The value or values described by this watch statement are shown in the watch window. The watch window is updated to show new values each time the value of the watch statement changes during program execution. Since the watch window does not exist in sequential mode, you must use the Watch List command to examine the values of watch statements.

When setting a watch expression, you can specify the format in which the value will be displayed. Type the expression followed by a comma and a format specifier. If you do not give a format specifier, the CodeView debugger displays the value in a default format. See Chapter 6, "Examining Data and Expressions," for more information about format specifiers and the default format.

Because the Microsoft Macro Assembler (Versions 1.0 through 4.0) does not write information to the object file on the type size of each variable, the Watch Expression command does not work with assembled programs. Use the Watch Memory command instead.

Note

If your program directly accesses absolute addresses used by IBM or IBM-compatible computers, you may sometimes get unexpected results with the Display Expression and Dump commands. However, the Watch command will usually show the correct values. This problem can arise if the CodeView debugger and your program try to use the same memory location.

This often occurs when a program reads data directly from the screen buffer of the display adapter. If you have an array called SCREEN that is initialized to the starting address of the screen buffer, the command DB SCREEN L 16 will show data from the CodeView display rather than from the display of the program you are debugging. The command WB SCREEN L 16 will show data from the program's display (provided screen swapping or screen flipping was specified at start-up). This happens because watch-statement values are updated during program execution, and any values read from the screen buffer will be taken from the output screen rather than from the debugging screen.

■ Mouse

To set a watch-expression statement using the mouse, point to Watch on the menu bar, press a mouse button and drag the highlight down to the Add Watch selection, then release the button. A dialog box appears, asking for the expression to be watched. Type the expression and press the ENTER key or a mouse button.

You cannot use the mouse version of the command to specify a range of memory to be watched, as you can with the dialog version.

■ Keyboard

To set a watch-expression statement with a keyboard command, press ALT-W to open the Watch menu, then press ALT-A to select Add Watch. A dialog box appears, asking for the expression to be watched. Type the expression and press the ENTER key.

You cannot use the keyboard version of the command to specify a range of memory to be watched, as you can with the dialog version.

■ Dialog

To set a watch-expression statement or watch-memory statement using a dialog command, enter a command line with the following syntax:

W? *expression*[[,*format*]] Watch expression
W[[*type*]] {*address* | *range*} Watch memory

An *expression* used with the Watch command can be a simple variable, or a more complicated expression using several variables and operators. The expression should be no longer than the width of the watch window. You can specify *format* using a CodeView format specifier. See Chapter 6, "Examining Data and Expressions" or Appendix A, "Command and Mode Summary," for more information.

When watching a memory location, *type* is a one-letter size specifier from the following list:

Specifier	Size
None	Default type
B	Byte
A	ASCII
I	Integer (signed decimal word)
U	Unsigned (unsigned decimal word)
W	Word
D	Double word
S	Short real
L	Long real
T	10-byte real

The default type used if no type size is specified is the last type used by a Dump, Enter, Watch Memory, or Tracepoint Memory command. If none of these commands has been used during the session, the default type is byte.

The data will be displayed in a format similar to that used by the Dump commands (see Chapter 6, "Examining Data and Expressions," for more information). The *range* can be any length, but only one line of data will be displayed in the watch window. If you do not specify an ending address for the range, the default range is one object.

■ Examples

The following dialog commands display three watch statements in the watch window:

```
W? I
```

The above example displays the current value of the variable I.

```
W? SUMX/I
```

The above example displays the value of the expression SUMX/I.

```
WB DATA(41) L 16
```

The above example displays the first 16 values at the address of the 41st element of array DATA. The dots to the right of the byte values in the watch window represent the corresponding ASCII values, which in this case are unprintable values displayed as dots. Unprintable values are less than 33 or greater than 126. ASCII values are shown only if the specified *type* is **B** (byte) or **A** (ASCII).

These commands produce the watch window in Figure 8.1.

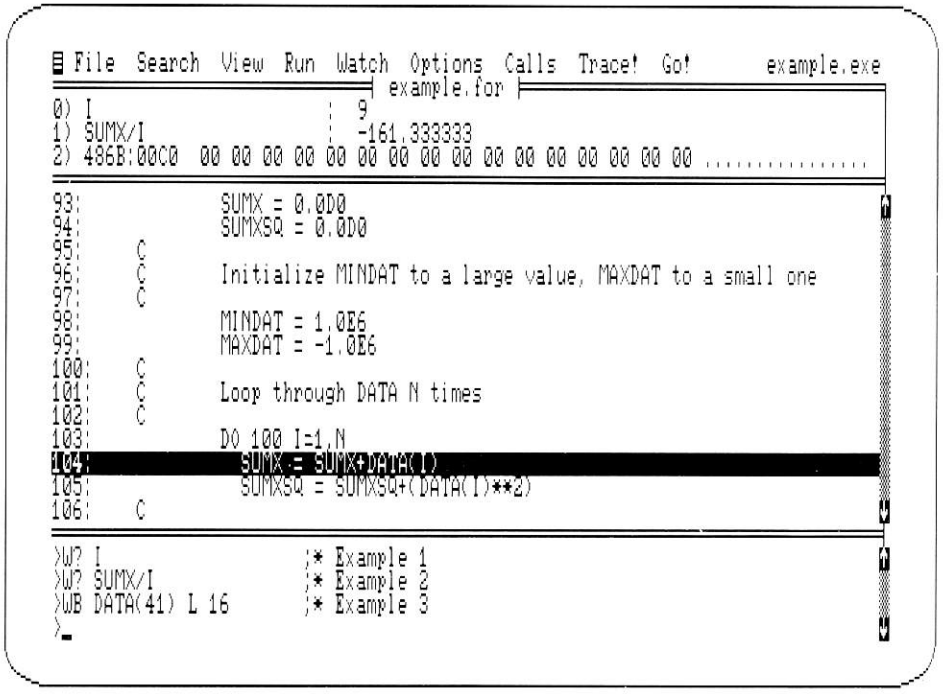
File Search View Run Watch Options Calls Trace! Go! example.exe
example.for
0) I : 9
1) SUMX/I : -161.333333
2) 486B:00C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
93: SUMX = 0.0D0
94: SUMXSQ = 0.0D0
95: C
96: C Initialize MINDAT to a large value, MAXDAT to a small one
97: C
98: MINDAT = 1.0E6
99: MAXDAT = -1.0E6
100: C
101: C Loop through DATA N times
102: C
103: DO 100 I=1,N
104: SUMX = SUMX+DATA(I)
105: SUMXSQ = SUMXSQ+(DATA(I)**2)
106: C
>W? I ;* Example 1
>W? SUMX/I ;* Example 2
>WB DATA(41) L 16 ;* Example 3
>

Figure 8.1 Watch-Command Statements in the Watch Window

8.3 Setting Watchpoints

The Watchpoint command is used to set a conditional breakpoint called a watchpoint. A watchpoint "watches" the value of an expression specified in a watch statement and breaks program execution when that value becomes true (nonzero). Note that the difference between watchpoints and tracepoints (described in Section 8.4) is that watchpoints break when the value of an expression changes from false to true, while tracepoints break whenever a value at a particular memory location changes.

A watch statement created by the Watchpoint command describes the expression that will be watched and compared to 0. The statement remains in memory until you delete it or quit the CodeView debugger. Any valid CodeView expression can be used as the watchpoint expression as long as the expression is not wider than the watch window.

In window mode, watchpoint statements and their values are displayed in high-intensity text in the watch window. In sequential mode, there is no watch window, so the values of watchpoint statements can only be displayed with the Watch List command (see Section 8.6 for more information).

Although a watchpoint expression can be any valid CodeView expression, the command works best with relational expressions that use the operators **.EQ.**, **.LT.**, **.LE.**, **.GT.**, **.GE.**, and **.NE.**. Relational expressions always evaluate to **.FALSE.** (0) or **.TRUE.** (1). Other expressions usually evaluate to values other than 0 or 1, thus the break is always taken.

■ Mouse

To set a watchpoint statement with the mouse, point to Watch on the menu bar, press a mouse button and drag the highlight down to the Watchpoint selection, then release the button. A dialog box appears, asking for the expression to be watched. Type the expression and press the ENTER key or a mouse button.

■ Keyboard

To execute the Watchpoint command with a keyboard command, press ALT-W to open the Watch menu, then press ALT-W to select Watchpoint. A dialog box appears, asking for the expression to be watched. Type the expression and press the ENTER key.

■ Dialog

To set a watchpoint using a dialog command, enter a command line with the following syntax:

WP? *expression*[[,*format*]]

The *expression* can be any valid CodeView expression (usually a relational expression). You can enter *format* as a CodeView format specifier, but there is little reason to do so, since the expression value is normally either 1 or 0.

■ Examples

The following dialog commands display two watch statements in the watch window:

```
WP? I .EQ. 5
```

The above example instructs the CodeView debugger to break when the variable I is equal to 5. After setting this watchpoint, you could use the Go command to execute until the condition becomes true.

```
WP? F .LT. 1
```

The above example instructs the CodeView debugger to break when the expression F becomes less than 1.

These commands produce the watch window in Figure 8.2.

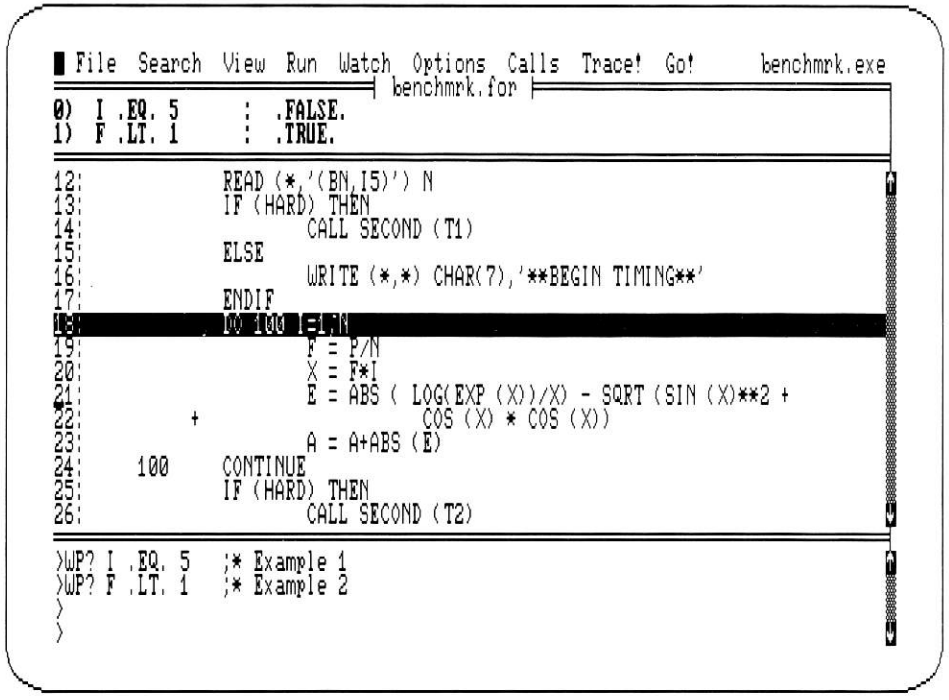
File Search View Run Watch Options Calls Trace! Go! benchmrk.exe
benchmrk.for
0) I .EQ. 5 : .FALSE.
1) F .LT. 1 : .TRUE.
READ (*,'(BN,I5)') N
IF (HARD) THEN
CALL SECOND (T1)
ELSE
WRITE (*,*) CHAR(7),'**BEGIN TIMING**'
ENDIF
DO 100 I=1,N
F = F/N
X = F*I
E = ABS (LOG(EXP (X))/X) - SQRT (SIN (X)**2 +
+ COS (X) * COS (X))
A = A+ABS (E)
100 CONTINUE
IF (HARD) THEN
CALL SECOND (T2)
)WP? I .EQ. 5 ;* Example 1
)WP? F .LT. 1 ;* Example 2

Figure 8.2 Watchpoint-Command Statements in the Watch Window

Note

Setting watchpoints significantly slows execution of the program being debugged. The CodeView debugger has to check to see if the expression is true each time a source line is executed in source mode, or each time an instruction is executed in assembly mode. Be careful when setting watchpoints near large or nested loops. A loop that executes almost instantly when run from DOS can take many minutes if executed from within the debugger with several watchpoints set.

Tracepoints do not slow CodeView execution as much as watchpoints, so you should use tracepoints when possible. For example, although you can set a watchpoint on a logical variable (`WP? MOVING`), a tracepoint on the same variable (`TP? MOVING`) has essentially the same effect and does not slow execution as much.

If you enter a seemingly endless loop, press CONTROL-BREAK or CONTROL-C to exit. You will soon learn the size of loop you can safely execute when watchpoints are set.

8.4 Setting Tracepoints

The Tracepoint command is used to set a conditional breakpoint called a tracepoint. A tracepoint breaks program execution when there is a change in the value of a specified expression or range of memory. Unlike the Watchpoint command, which checks the values of expressions, Tracepoint watches memory locations. Tracepoint breaks program execution whenever that value changes, whereas Watchpoint breaks only when the expression value becomes true.

The watch statement created by the Tracepoint command describes the expression or memory range to be watched and tested for change. The statement remains in memory until you delete it or quit the CodeView debugger.

An expression used with the Tracepoint command must refer to a memory location of not more than 128 bytes. For example, `I .EQ. 10` would be invalid because it is either 1 (true) or 0 (false), rather than a value stored in memory. The expression `SYM1+SYM2` is invalid because it is the calculated sum of the value of two memory locations.

Arrays must have subscripts when used with Tracepoint. The expression ARRAY is invalid. However, if the array is declared as CHARACTER ARRAY(64), when the Tracepoint command is given the argument ARRAY(32) one element of ARRAY (the 32nd) is checked.

In window mode, tracepoint statements and their values are shown in high-intensity text in the watch window. In sequential mode, there is no watch window, so the values of tracepoint statements can only be displayed with the Watch List command (see Section 8.6, "Listing Watchpoints and Tracepoints," for more information).

When setting a tracepoint expression, you can specify the format in which the value will be displayed. Type the expression followed by a comma and a format specifier. If you do not give a format specifier, the CodeView debugger displays the value in a default format. See Chapter 6, "Examining Data and Expressions," for more information about format specifiers and the default format.

The expression version of the Tracepoint command does not work for programs assembled with Microsoft Macro Assembler versions 4.0 and earlier, because the assembler does not write information to the object file about the type size of each variable. Use the memory version instead.

■ Mouse

To set a tracepoint-expression statement with the mouse, point to Watch on the menu bar, press a mouse button and drag the highlight down to the Tracepoint selection, then release the button. A dialog box appears, asking for the expression to be watched. Type the expression and press the ENTER key or a mouse button.

You cannot specify a range of memory to be watched with the mouse version of the command, as you can with the dialog version.

■ Keyboard

To set a tracepoint-expression statement with a keyboard command, press ALT-W to open the Watch menu, then press ALT-T to select Tracepoint. A dialog box appears, asking for the expression to be watched. Type the expression and press the ENTER key.

You cannot use the keyboard version of the command to specify a range of memory to be watched, as you can with the dialog version.

■ Dialog

To set a tracepoint using a dialog command, enter a command line with the following syntax:

TP? *expression*[[,*format*]]	Expression tracepoint
TP[[*type*]]{*address* | *range*}	Memory tracepoint

An *expression* used with the Tracepoint command can be a simple variable or a more complicated expression using several variables and operators. The expression should not be longer than the width of the watch window. You can specify *format* using a CodeView format specifier if you do not want the value to be displayed in the default format (decimal for integers or floating point for real numbers). See Section 6.2, "Display Expression Command," for more information.

In the memory-tracepoint form, *range* must be a valid address range and *type* must be a one-letter memory-size specifier. If you specify only the start of the range, the CodeView debugger displays one object as the default.

Although no more than one line of data will be displayed in the watch window, the range to be checked for change can be any size up to 128 bytes. The data will be displayed in the format used by the Dump commands (see Chapter 6, "Examining Data and Expressions," for more information). The valid memory-size specifiers are listed below:

Specifier	Size
None	Default type
B	Byte
A	ASCII
I	Integer (signed decimal word)
U	Unsigned (unsigned decimal word)
W	Word
D	Double word
S	Short real
L	Long real
T	10-byte real

The default type if no type size is specified is the last type used by a Dump, Enter, Watch Memory, or Tracepoint Memory command. If none of these commands has been used during the session, the default type is byte.

■ Examples

The following dialog commands display two watch statements in the watch window:

```
TP? N
```

The above example instructs the CodeView debugger to stop whenever the value of the variable I changes.

```
TP? DATA(2)
```

The above example instructs the debugger to stop whenever there is a change in the value at the address of the array element DATA(2).

These commands produce the watch window in Figure 8.3.

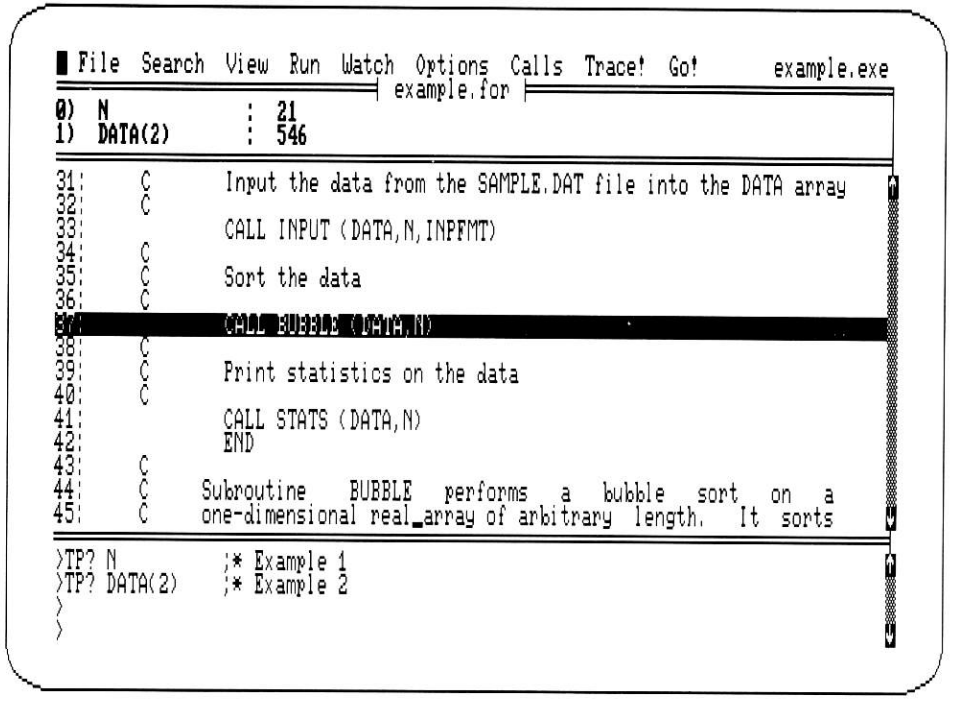
File Search View Run Watch Options Calls Trace! Go! example.exe
example.for
0) N : 21
1) DATA(2) : 546
31: C Input the data from the SAMPLE.DAT file into the DATA array
32: C
33: CALL INPUT (DATA,N,INPFMT)
34: C
35: C Sort the data
36: C
37: CALL BUBBLE (DATA,N)
38: C
39: C Print statistics on the data
40: C
41: CALL STATS (DATA,N)
42: END
43: C
44: C Subroutine BUBBLE performs a bubble sort on a
45: C one-dimensional real_array of arbitrary length. It sorts
>TP? N ;* Example 1
>TP? DATA(2) ;* Example 2
>

Figure 8.3 Tracepoints in the Watch Window

Note

Setting tracepoints significantly slows execution of the program being debugged. The CodeView debugger has to check to see if the expression or memory range has changed each time a source line is executed in source mode or each time an instruction is executed in assembly mode. However, because tracepoints do not slow execution as much as watchpoints, you should use tracepoints whenever possible. Be careful when setting tracepoints near large or nested loops. A loop that executes almost instantly when run from DOS can take many minutes if executed from within the debugger with several tracepoints set.

If you enter a seemingly endless loop, press CONTROL-BREAK or CONTROL-C to exit. Often you can tell how far you went in the loop by the value of the tracepoint when you exited.

8.5 Deleting Watch Statements

The Watch Delete command enables you to delete watch statements that were set previously with the Watch, Watchpoint, or Tracepoint command.

When you delete a watch statement in window mode, the statement disappears and the watch window closes around it. For example, if there are three watch statements in the window and you delete Statement 1, the window is redrawn with one less line. Statement 0 remains unchanged, but Statement 2 becomes Statement 1. If there is only one statement, the window disappears when that statement is deleted.

■ Mouse

To delete a watch statement with the mouse, point to Watch on the menu bar, press a mouse button and drag the highlight down to the Delete Watch selection, then release the button. A dialog box appears, containing all the watch statements. Point to the statement you want to delete and press the ENTER key or a mouse button. The dialog box disappears and the watch window is redrawn without the deleted watch statement.

■ Keyboard

To execute the Execute command with a keyboard command, press ALT-W to open the Watch menu, then press ALT-D to select Delete Watch. A dialog box appears, containing all the watch statements. Use the UP ARROW and DOWN ARROW keys to move the cursor to the statement you want to delete, then press the ENTER key. The dialog box disappears and the watch window is redrawn without the deleted watch statement.

■ Dialog

To delete watch statements using a dialog command, enter a command line with the following syntax:

Y *number*

When you set a watch statement, it is automatically assigned a number (starting with 0). In window mode, the number appears to the left of the watch statement in the watch window. In sequential mode, you can use the Watch List (**W**) command to view the numbers of current watch statements.

You can delete existing watch statements by specifying the *number* of the statement you want to delete with the Delete Watch command. (The **Y** is a mnemonic for Yank.)

You can use the asterisk (*) to represent all watch statements.

■ Examples

```
>Y 2
>
```

The above example deletes Watch statement 2.

```
>Y *
>
```

The above example deletes all watch statements and closes the watch window.

8.6 Listing Watchpoints and Tracepoints

The Watch List command lists all previously set watchpoints and tracepoints with their assigned numbers and their current values.

This command is the only way to examine current watch statements in sequential mode. The command has little use in window mode, since watch statements are already visible in the watch window.

■ Mouse

The Watch List command cannot be executed with the mouse.

■ Keyboard

The Watch List command cannot be executed with a keyboard command.

■ Dialog

To list watch statements using a dialog command, enter a command line with the following syntax:

W

The display is the same as the display that appears in the watch window in window mode.

Note

The command letter (**W**) for the Watch List command is the same as the command letter for the memory version of the Watch command when no memory size is given. The difference between the commands is that the Watch List command never takes an argument. The Watch command always requires at least one argument.

■ Example

```
>W
0) N   :  40
1) SUMX/I : -161.333333
2) 486B:00C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
>
```

Chapter 9
Examining Code

9.1 Introduction 161
9.2 Set Mode Command 161
9.3 Unassemble Command 163
9.4 View Command 166
9.5 Current Location Command 169
9.6 Stack Trace Command 170

9.1 Introduction

Several CodeView commands allow you to examine program code, or data related to code. The following commands are discussed in this chapter:

Command	Action
Set Mode (**S**)	Sets format for code displays
Unassemble (**U**)	Displays assembly instructions
View (**V**)	Displays source lines
Current Location (.)	Displays the current location line
Stack Trace (**K**)	Displays routines or procedures

9.2 Set Mode Command

The Set Mode command sets the mode in which code is displayed. The two basic display modes are source mode, in which the program is displayed as source lines, and assembly mode, in which the program is displayed as assembly-language instructions. These two modes can be combined in mixed mode, in which the program is displayed with both source lines and assembly-language instructions.

In sequential mode, there are three display modes: source, assembly, and mixed. These modes affect the output of commands that display code (Register, Trace, Program Step, Go, Execute, and Unassemble).

In window mode, there are two display modes: source and assembly, but there are additional options for mixing source and assembly modes and controlling the display of assembly-language instructions. The display mode affects the way the program is shown in the display window.

Source and mixed modes are only available if the executable file contains symbols in the CodeView format. Programs that do not contain symbolic information (including all **.COM** files) are displayed in assembly mode.

■ Mouse

To set the display mode with the mouse, point to View on the menu bar, press a mouse button and drag the highlight to either the Source selection for source mode or the Assembly selection for assembly mode. Then release the button.

You can further control the display of assembly-language instructions by making selections from the Options menu. See Section 3.2.3.6, "Using the Options Menu," for more information.

■ Keyboard

To change the display mode with a keyboard command, press the F3 key. If the current mode is source, the new mode will be assembly. If the current mode is assembly, the new mode will be source. This command works in either window or sequential mode. In sequential mode, the word `source` or `assembly` is displayed to indicate the new mode.

In window mode you can further control the display of assembly-language instructions by making selections from the Options menu. See Section 3.2.3.6, "Using the Options Menu," for more information.

■ Dialog

To set the display mode from the dialog window, enter a command line with the following syntax:

S[[**+** | **−** | **&**]]

If the plus sign is specified (**S+**), source mode is selected, and the word `source` is displayed.

If the minus sign is specified (**S−**), assembly mode is selected, and the word `assembly` is displayed. In window mode, the display will include any assembly options, except the Mix Source option, previously toggled on from the Options menu. The Mix Source option is always turned off by the **S−** command.

If the ampersand is specified (**S&**), mixed mode is selected, and the word `mixed` is displayed. In window mode, the display includes any assembly options previously toggled on from the Options menu. In addition, the Mixed Source option will be turned on by the **S&** command.

If no argument is specified (**S**), the current mode (`source`, `assembly`, or `mixed`) is displayed.

The Unassemble command in sequential mode is an exception in that it displays mixed source and assembly with both the source (**S+**) and mixed (**S&**) modes. When you enter the dialog version of the Set Mode command, the CodeView debugger outputs the name of the new display mode: `source`, `assembly`, or `mixed`.

Note

The 80286 protected-mode mnemonics cannot be displayed in assembly or mixed mode. They will not be shown in the display window in window mode.

■ Examples

```
>S+
source
>S-
assembly
>S&
mixed
>
```

The above examples show the source mode being changed to `source`, `assembly`, and `mixed`. In window mode, the commands change the format of the display window. In sequential mode, the commands change the output from the commands that display code (Register, Trace, Program Step, Go, Execute, and Unassemble). See the sections on individual commands for examples of how they are affected by the display mode.

9.3 Unassemble Command

The Unassemble command displays the assembly-language instructions of the program being debugged. It is most useful in sequential mode, where it is the only method of examining a sequence of assembly-language instructions. In window mode it can be used to display a specific portion of assembly-language code in the display window.

Note

Occasionally, code similar to the following will be displayed:

```
FE30    ???    Byte Ptr  [BX + SI]
```

These are instructions Intel recommends against using because equivalent opcodes exist and the instruction may not work properly on all machines, or because the opcodes are reserved. You should not see this type of code unless you are unassembling data.

■ Mouse

The Unassemble command has no direct mouse equivalent, but you can view unassembled code at any time by changing the mode to assembly or mixed (see Section 9.2, "Set Mode Command," for more information).

■ Keyboard

The Unassemble command has no direct keyboard equivalent, but you can view unassembled code at any time by changing the mode to assembly or mixed (see Section 9.2, "Set Mode Command," for more information).

■ Dialog

To display unassembled code using a dialog command, enter a command line with the following syntax:

U [[*address* | *range*]]

The effect of the command varies depending on whether you are in sequential or window mode.

In sequential mode, if you do not specify *address* or *range*, the disassembled code begins at the current unassemble address and shows the next eight lines of instructions. The unassemble address is the address of the instruction after the last instruction displayed by the previous Unassemble command. If the Unassemble command has not been used during the session, the unassemble address is the current instruction.

If you specify an *address*, the disassembly starts at that address and shows the next eight lines of instructions. If you specify a *range*, the instructions within the range will be displayed.

The format of the display depends on the current display mode (see Section 9.2, "Set Mode Command," for more information). If the mode is source (**S+**) or mixed (**S&**), the CodeView debugger displays source lines mixed with unassembled instructions. One source line is shown for each corresponding group of assembly-language instructions. If the display mode is assembly, only assembly-language instructions are shown.

In window mode, the Unassemble command changes the mode of the display window to assembly. The display format will reflect any options previously set from the Options menu. There is no output to the dialog window. If *address* is given, the instructions in the display window will begin at the specified address. If *range* is given, only the starting address will be used. If no argument is given, the debugger scrolls down and displays the next screen of assembly-language instructions.

Note

80286 protected-mode mnemonics cannot be displayed with the Unassemble command.

■ Examples

```
>S&
mixed
>U 0x11
49D0:0011 35068E        XOR   AX,__sqrtjmptab+8cd4 (8E06)
49D0:0014 189A2300      SBB   Byte Ptr [BP+SI+0023],BL
49D0:0018 FC            CLD
49D0:0019 49            DEC   CX
49D0:001A CD351ED418    INT   35 ;FSTP    DWord Ptr [__fpinit+ee (18D4)]
49D0:001F CD3D          INT   3D ;FWAIT
7:                A = 0.0
49D0:0021 CD35EE        INT   35 ;FLDZ
```

The above example sets the mode to mixed and unassembles eight lines of disassembled code, plus whatever source lines are encountered within those lines. The display would be the same if the mode were source.

```
>S-
assembly
>U 0x11
49D0:0011 35068E        XOR   AX,__sqrtjmptab+8cd4 (8E06)
49D0:0014 189A2300      SBB   Byte Ptr [BP+SI+0023],BL
49D0:0018 FC            CLD
49D0:0019 49            DEC   CX
49D0:001A CD351ED418    INT   35 ;FSTP    DWord Ptr [__fpinit+ee (18D4)]
49D0:001F CD3D          INT   3D ;FWAIT
49D0:0021 CD35EE        INT   35 ;FLDZ
>
```

The example shown above sets the mode to assembly and repeats the same command.

9.4 View Command

The View command displays the lines of a text file (usually a source module or include file). It is most useful in sequential mode, where it is the only method of examining a sequence of source lines. In window mode, the View command can be used to page through the source file or to load a new source file.

■ Mouse

To load a new source file with the button, point to File on the menu bar, press a mouse button and drag the highlight to the Load selection, then release the button. A dialog box appears, asking for the name of the file you wish to load. Type the name of the file, and press the ENTER key or a mouse button. The new file appears in the display window.

The paging capabilities of the View command have no direct mouse equivalent, but you can move about in the source file by pointing to the up or down arrows on the scroll bars and then clicking different mouse buttons. See Chapter 3, "The CodeView Display," for more information about paging with the mouse.

■ Keyboard

To load a new source file with a keyboard command, press ALT-F to open the File menu, then press ALT-L to select Load. A dialog box appears, asking for the name of the file you wish to load. Type the name of the file, and press the ENTER key. The new file appears in the display window.

The paging capabilities of the View command have no direct keyboard equivalent, but you can move about in the source file by first putting the cursor in the display window with the F6 key, then pressing the PGUP, PGDN, HOME, END, and UP ARROW and DOWN ARROW keys. See Chapter 3, "The CodeView Display," for more information about paging with keyboard commands.

■ Dialog

To display source lines using a dialog command, enter a command line with the following syntax:

V [[*expression*]]

Since addresses for the View command are often specified as a line number (with an optional source file), a more specific syntax for the command would be as follows:

V [[**.**[[*filename*:]]*linenumber*]]

The effect of the command varies, depending on whether you are in sequential or window mode.

In sequential mode, the View command displays eight source lines. The starting source line is one of the following:

- The current source line if no argument is given.
- The specified *linenumber*. If *filename* is given, the specified file is loaded, and the *linenumber* refers to lines in it.
- The address that *expression* evaluates to. For example, *expression* could be a procedure name or an address in the *segment*:*offset* format. The code segment is assumed if no segment is given.

In sequential mode, the View command is not affected by the current display mode (source, assembly, or mixed); source lines are displayed regardless of the mode.

In window mode, if you enter the View command while the display mode is assembly, the CodeView debugger will automatically switch back to source mode. If you give *linenumber* or *expression*, the display window will be redrawn so that the source line corresponding to the given *address* will appear at the top of the source window. If you specify a *filename* with a *linenumber*, the specified file will be loaded.

If you enter the View command with no arguments, the display will scroll down one line short of a page; that is, the source line that was at the bottom of the window will be at the top.

Note

The View command with no argument is similar to pressing the PGDN key, or clicking right on the down arrow with the mouse. The difference is that pressing the PGDN key enables you to scroll down one more line.

■ Examples

```
>V BUBBLE
51:             IF (N .LE. 1) GOTO 101
52:             DO 201 I = 1,N-1
53:             DO 301 J = I + 1,N
54:             IF (X(I) .LE. X(J)) GOTO 301
55:             TEMP = X(I)
56:             X(I) = X(J)
57:             X(J) = TEMP
58:         301 CONTINUE
```

The above example (shown in sequential mode) displays eight source lines, beginning at routine BUBBLE.

```
>V .samp3.for:2
2:       C        Subroutine STATS - Print minimum,
3:       C        maximum, average, and standard
4:       C        standard deviation of DATA array
5:       C
6:                SUBROUTINE STATS (DATA,N)
7:                IMPLICIT INTEGER*2 (I-N)
8:                REAL DATA(N)*4,SUMX*8,SUMXSQ*8,MAX,MIN
```

The above example loads the source file SAMP3.FOR and displays seven source lines starting at line 2.

9.5 Current Location Command

The Current Location command displays the source line or assembly-language instruction corresponding to the current program location.

■ Mouse

The Current Location command cannot be executed with the mouse.

■ Keyboard

The Current Location command cannot be executed with a keyboard command.

■ Dialog

To display the current location line using a dialog command, enter a command line with the following syntax:

.

In sequential mode, the command displays the current source line. The line is displayed regardless of whether the current debugging mode is source or assembly. If the program being debugged has no symbolic information, the command will be ignored.

In window mode, the command puts the current program location (marked with reverse video or a contrasting color) in the center of the display window. The display mode (source or assembly) will not be affected. This command is useful if you have scrolled through the source code or assembly-language instructions so that the current location line is no longer visible.

For example, if you are in window mode and have executed the program being debugged to somewhere near the start of the program, but you have scrolled the display to a point near the end, the Current Location command returns the display to the current program location.

- **Example**

```
>.
MINDAT = 1.0E6
>
```

The above example illustrates how to display the current source line in sequential mode. The same command in window mode would not produce any output, but it could change the text shown in the display window.

9.6 Stack Trace Command

The Stack Trace command allows you to display routines that have been called during program execution. The first line of the display shows the name of the current routine. The succeeding lines (if any) list any other routines that were called to reach the current address.

For each routine, the values of any arguments are shown in parentheses after the routine name. Values are shown in the current radix (the default is decimal).

The term "stack trace" is used because, as each routine is called, its address and arguments are stored (pushed) on the program stack. Therefore, tracing through the stack shows the currently active routines. For FORTRAN programs, the **main** routine will always be at the bottom of the stack. In assembly-language programs, the last routine displayed in the stack trace is **astart** instead of **main**.

The Stack Trace command also enables you to find and view the source lines where individual routines were called.

Note

This discussion refers to routines, since that is the terminology for FORTRAN programs. In assembly mode, the term "procedure" may be more accurate. If you are using the CodeView debugger to debug assembly-language programs, the Stack Trace command will work only if procedures were called with the calling convention used by Microsoft languages. This calling convention is explained in Chapter 11 of the *Microsoft FORTRAN Compiler User's Guide.*

■ Mouse

To view a stack trace with the mouse, point to Calls on the menu bar and press a mouse button. The Calls menu will appear, showing the current routine at the top and other routines below it in the reverse order in which they were called; for example, the first routine called will be at the bottom. The values of any routine arguments will be shown in parentheses following the routines.

If you want to view code at the point where one of the routines was called, hold the mouse button down and drag the highlight to the routine below the one you want to view, then release the button. The cursor will move to the calling source line (in source mode) or the calling instruction (in assembly or mixed mode). In other words, the cursor will indicate the calling location in the selected routine where the next-level routine was called. If you select the current (top-level) routine, the cursor moves to the current location in that routine.

■ Keyboard

To view a stack trace with a keyboard command, press ALT-C to open the Calls menu. The menu will show the current routine at the top, and other routines below it in the reverse order in which they were called; for example, the first routine called will be at the bottom. The values of any routine arguments will be shown in parentheses following the routine.

If you want to view code at the point where one of the routines was called, press the down arrow to move the highlight down to the routine below the one you want to view, then press the ENTER key. The cursor will move to the calling source line (in source mode) or the calling instruction (in assembly or mixed mode). In other words, the cursor will indicate the calling location in the selected routine where the next-level routine was called. If you select the current (top-level) routine, the cursor moves to the current location in that routine.

■ Dialog

To display a stack trace with a dialog command, enter a command line with the following syntax:

K

The output from the Stack Trace dialog command lists the routines in the reverse order in which they were called. The arguments to each routine are

shown in parentheses. Finally, the line number from which the routine was called is shown.

You can enter the line number as an argument to the View or Unassemble command if you want to view code at the point where the routine was called.

In window mode, the output from the Stack Trace dialog command appears in the dialog window. You may need the dialog version rather than the menu version, since the Calls menu can be truncated if there are too many routines or routine arguments. The dialog display wraps around if necessary, so that you can see all routines and all arguments.

■ Example

```
>K
ANALYZE(67,0), line 94
COUNTWORDS(0,512), line 73
MAIN(2,5098), line 42
>
```

In the above example, the first line of output indicates that the current routine is ANALYZE. Its first argument currently has a decimal value of 67, and its second argument has a value of 0. The current location in this routine is line 94.

The second line indicates that ANALYZE was called by COUNTWORDS, and that its arguments have the values 0 and 512. Routine ANALYZE was called from line 73 of routine COUNTWORDS.

Likewise, COUNTWORDS was called from line 42 of MAIN, and its arguments have the values 2 and 5098.

If the radix had been set to 16 or 8 using the Radix (**N**) command, the arguments would be shown in that radix. For example, the last line would be shown as MAIN(2,13ea) in hexadecimal or MAIN(2,11752) in octal.

Chapter 10
Modifying Code or Data

10.1 Introduction 175
10.2 Assemble Command 175
10.3 Enter Commands 178
10.3.1 Enter Command 182
10.3.2 Enter Bytes Command 183
10.3.3 Enter ASCII Command 183
10.3.4 Enter Integers Command 184
10.3.5 Enter Unsigned Integers Command 185
10.3.6 Enter Words Command 186
10.3.7 Enter Double Words Command 186
10.3.8 Enter Short Reals Command 187
10.3.9 Enter Long Reals Command 188
10.3.10 Enter 10-Byte Reals Command 189
10.4 Register Command 190

10.1 Introduction

The CodeView debugger provides the following commands for modifying code or data in memory:

Command	Action
Assemble (**A**)	Modifies code
Enter (**E**)	Modifies memory, usually data
Register (**R**)	Modifies registers and flags

Changes to code are temporary. You can use them for testing in the CodeView debugger, but you cannot save them or permanently change the program. To make permanent changes, you must modify the source code and recompile.

10.2 Assemble Command

The Assemble command assembles 8086-family (8086, 8087, 8088, 80186, 80287, and 80286 unprotected) instruction mnemonics and places the resulting instruction code into memory at a specified address. The only 8086-family mnemonics that cannot be assembled are 80286 protected-mode mnemonics.

■ Mouse

The Assemble command cannot be executed with the mouse.

■ Keyboard

The Assemble command cannot be executed with a keyboard command.

■ Dialog

To assemble code using a dialog command, enter a command line with the following syntax:

A[[*address*]]

If *address* is specified, the assembly starts at that address; otherwise the current assembly address is assumed.

The assembly address is normally the current address (the address pointed to by the **CS** and **IP** registers). However, when you use the Assemble command, the assembly address is set to the address immediately following the last instruction where you assembled an instruction. When you enter any command that executes code (Trace, Program Step, Go, or Execute), the assembly address is reset to the current address.

When you type the Assemble command, the assembly address is displayed. The CodeView debugger then waits for you to enter a new instruction in the standard 8086-family instruction-mnemonic form. You can enter instructions in uppercase, lowercase, or both.

To assemble a new instruction, type the desired mnemonic and press the ENTER key. The debugger assembles the instruction into memory and displays the next available address. Continue entering new instructions until you have assembled all the instructions you want. To conclude assembly and return to the CodeView prompt, press the ENTER key only.

If an instruction you enter contains a syntax error, the debugger displays the message `^ Syntax error`, redisplays the current assembly address, and waits for you to enter a correct instruction. The caret symbol in the message will point to the first character the CodeView debugger could not interpret.

The following nine rules govern entry of instruction mnemonics:

1. The far-return mnemonic is **RETF**.

2. String mnemonics must explicitly state the string size. For example, use **MOVSW** to move word strings and **MOVSB** to move byte strings.

3. The CodeView debugger automatically assembles short, near, or far jumps and calls, depending on byte displacement to the destination address. These may be overridden with the **NEAR** or **FAR** prefix, as shown in the following examples:

```
JMP     #502
JMP     NEAR #505
JMP     FAR  #50A
```

   The **NEAR** prefix can be abbreviated to **NE**, but the **FAR** prefix cannot be abbreviated.

4. The CodeView debugger cannot determine whether some operands refer to a word memory location or to a byte memory location. In these cases, the data type must be explicitly stated with the prefix **WORD PTR** or **BYTE PTR**. Acceptable abbreviations are **WO** and **BY**. Examples are shown below:

```
MOV     WORD PTR [BP],1
MOV     BYTE PTR [SI-1],symbol
MOV     WO PTR [BP],1
MOV     BY PTR [SI-1],symbol
```

5. The CodeView debugger cannot determine whether an operand refers to a memory location or to an immediate operand. The debugger uses the convention that operands enclosed in square brackets refer to memory. Two examples are shown below:

```
MOV     AX,#21
MOV     AX,[#21]
```

   The first statement moves `#21` into `AX`. The second statement moves the data at offset `#21` into `AX`.

6. The **DB** instruction assembles byte values directly into memory. The **DW** instruction assembles word values directly into memory, as shown in the following examples:

```
DB      1,2,3,4,"This is an example."
DW      1000,2000,3000,"Bach"
```

7. The CodeView debugger supports all forms of indirect register instructions, as shown in the following examples:

```
ADD     BX,[BP+2].[SI-1]
POP     [BP+DI]
PUSH    [SI]
```

8. All instruction-name synonyms are supported, as shown in the following examples:

```
LOOPZ    #100
LOOPE    #100
JA       #200
JNBE     #200
```

   If you assemble instructions and then examine them with the Unassemble command (**U**), the CodeView debugger may show synonymous instructions, rather than the ones you assembled.

9. Do not assemble and execute 8087 or 80287 instructions if your system is not equipped with one of these math coprocessor chips. The **WAIT** instruction, for example, will cause your system to hang up if you try to execute it without the appropriate chip.

■ **Example**

```
>U #40 L 1
39B0:0040 89C3              MOV         BX,AX
>A #40
39B0:0040 MOV       CX,AX
39B0:0042
>U #40 L 1
39B0:0040 89C1              MOV         CX,AX
>
```

The above example modifies the instruction at address #40 so that it moves data into the CX register instead of the BX register. The Unassemble command (**U**) is used to show the instruction before and after the assembly.

You can modify a portion of code for testing, as in the example, but you cannot save the modified program. You must modify your source code and recompile.

10.3 Enter Commands

The CodeView debugger has several commands for entering data to memory. You can use these commands to modify either code or data, though code can usually be modified more easily with the Assemble command (**A**). The Enter commands are listed below:

Command	Command Name
E	Enter (size is the default type)
EB	Enter Bytes
EA	Enter ASCII
EI	Enter Integers
EU	Enter Unsigned Integers
EW	Enter Words
ED	Enter Double Words
ES	Enter Short Reals
EL	Enter Long Reals
ET	Enter 10-Byte Reals

■ Mouse

The Enter commands cannot be executed with the mouse.

■ Keyboard

The Enter commands cannot be executed with keyboard commands.

■ Dialog

To enter data to memory with a dialog command, enter a command line with the following syntax:

E[[*type*]] *address* [[*list*]]

The *type* is a one-letter specifier that indicates the type of the data to be entered. The *address* indicates where the data will be entered. If no segment is given in the address, the data segment (**DS**) is assumed.

The *list* can consist of one or more expressions that evaluate to data of the size specified by *type*. This data will be entered to memory at *address*. If one of the values in the list is invalid, an error message will be displayed. The values preceding the error are entered; values at and following the error are not entered.

The expressions in the list are evaluated in the current radix, regardless of the size and type of data being entered. For example, if the radix is 10 and you give the value 10 in a list with the Enter Words command, the decimal value 10 will be entered even though word values are normally entered in hexadecimal. This means that the Enter Words, Enter Integers, and Enter Unsigned Integers commands are identical when used with the list method, since 2-byte data is being entered for each command.

If *list* is not given, the CodeView debugger will prompt for values to be entered to memory. Values entered in response to prompts are accepted in hexadecimal for the Enter Bytes, Enter ASCII, Enter Words, and Enter Double Words commands. The Enter Integers command accepts signed decimal integers, while the Enter Unsigned Integers command accepts unsigned decimal integers. The Enter Short Reals, Enter Long Reals, and Enter 10-Byte Reals commands accept decimal floating-point values.

With the prompting method of data entry, the CodeView debugger prompts for a new value at *address* by displaying the address and its current value. You can then replace the value, skip to the next value, return to a previous value, or exit the command, as explained below:

- To replace the value, type the new value after the current value.
- To skip to the next value, press the SPACEBAR. Once you have skipped to the next value, you can change its value or skip to the following value. If you pass the end of the display, the CodeView debugger displays a new address to start a new display line.
- To return to the preceding value, type a backslash (\). When you return to the preceding value, the debugger starts a new display line with the address and value.
- To stop entering values and return to the CodeView prompt, press the ENTER key. You can exit the command at any time.

Sections 10.3.1 – 10.3.10 discuss the Enter commands in order of the size of data they accept.

■ Examples

```
>EW PLACE 16 32
```

The above example shows how to enter two word-sized values at the address PLACE. If the default radix (decimal) is in effect, this command enters the hexadecimal values #10 and #20.

```
>EW PLACE

3DA5:0B20  00F3._
```

The above example illustrates the prompting method of entering data. When you supply the address where you want to enter data but supply no data to be entered there, the CodeView debugger displays the current value of the address and waits for you to enter a new value. The underscore in these examples represents the CodeView cursor. You could change the value F3 to the new value 16 (#10) by typing 1 0 (but don't press the ENTER key yet). The value must be typed in hexadecimal for the Enter Words command, as shown below:

```
>EW PLACE

3DA5:0B20  00F3.10_
```

You could then skip to the next value by pressing the SPACEBAR. The CodeView debugger responds by displaying the next value, as shown below:

```
>EW PLACE

3DA5:0B20  00F3.10  4F20._
```

You could then type another hexadecimal value, such as 30:

```
>EW PLACE

3DA5:0B20  00F3.10  4F20.30_
```

Press the SPACEBAR to move to the next value:

```
>EW PLACE

3DA5:0B20  00F3.10  4F20.30  3DC1._
```

Assume you realize that the last value entered, 30, is incorrect. You really wanted to enter 20. You could return to the previous value by typing a backslash. The CodeView debugger starts a new line, starting with the previous value. Note that the backslash is not echoed:

```
>EW PLACE

3DA5:0B20  00F3.10  4F20.30  3DC1.
3DA5:0B22  0030._
```

Type the correct value, 20:

```
>EW PLACE

3DA5:0B20   00F3.10   4F20.30   3DC1.
3DA5:0B22   0030.20_
```

If this is the last value you want to enter, press the ENTER key to stop. The CodeView prompt reappears, as shown below:

```
>EW PLACE

3DA5:0B20   00F3.10   4F20.30   3DC1.
3DA5:0B22   0030.20
>_
```

10.3.1 Enter Command

- **Syntax**

E *address* [[*list*]]

The Enter command enters one or more values into memory at the specified *address*. The data is entered in the format of the default type, which is the last type specified with a Dump, Enter, Watch Memory, or Tracepoint Memory command. If none of these commands has been entered during the session, the default type is bytes.

Use this command with caution when entering values in the list format; values will be truncated if you enter a word-sized value when the default type is actually bytes. If you are not sure of the current default type, specify the size in the command.

Important

The Execute command and the Enter command have the same command letter (**E**). The difference is that the Execute command never takes an argument; the Enter command always requires at least one argument.

10.3.2 Enter Bytes Command

■ Syntax

EB *address* [[*list*]]

The Enter Bytes command enters one or more byte values into memory at *address*. The optional *list* can be entered as a list of expressions separated by spaces. The expressions are evaluated and entered in the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in hexadecimal.

The Enter Bytes command can also be used to enter strings, as described in Section 10.3.3, "Enter ASCII Command."

■ Examples

```
>EB #100 10 20 30
>
```

If the current radix is 10, the above example replaces the three bytes at DS:#100, DS:#101, and DS:#102 with the decimal values 10, 20, and 30.

```
>EB #100

3DA5:0100  130F.A
>
```

The above example replaces the byte at DS:#100 with 10 (#A).

10.3.3 Enter ASCII Command

■ Syntax

EA *address* [[*list*]]

The Enter ASCII command works in the same way as the Enter Bytes command (**EB**) described in Section 10.3.2. The *list* version of this command can be used to enter a string expression. You can include escape sequences in strings.

■ Example

```
>EA message "Can\'t open file"
>
```

In the example above, the string `Can\'t open file` is entered starting at the symbolic address `message`. (Note that the double quotation marks are CodeView string delimiters. FORTRAN string delimiters are single quotation marks.)

You can also use the Enter Bytes command to enter a string expression, or you can enter nonstring values using the Enter ASCII command.

10.3.4 Enter Integers Command

■ Syntax

EI *address* [[*list*]]

The Enter Integers command enters one or more word values into memory at *address* using the signed-integers format. With the CodeView debugger, a signed integer can be any decimal integer between −32,768 and 32,767.

The optional *list* can be entered as a list of expressions separated by spaces. The expressions are entered and evaluated in the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in decimal.

■ Examples

```
>EI #100 -10 10 -20
>
```

If the current radix is 10, the above example replaces the three integers at DS:#100, DS:#102, and DS:#104 with the decimal values −10, 10, and −20.

```
>EI #100

3DA5:0100  130F.-10
>
```

The above example replaces the integer at DS:#100 with −10.

10.3.5 Enter Unsigned Integers Command

■ Syntax

EU *address* [[*list*]]

The Enter Unsigned Integers command enters one or more word values into memory at *address* using the unsigned-integers format. With the CodeView debugger, an unsigned integer can be any decimal integer between 0 and 65,535.

The optional *list* can be entered as a list of expressions separated by spaces. The expressions are entered and evaluated in the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in decimal.

■ Examples

```
>EU #100 10 20 30
>
```

If the current radix is 10, the above example replaces the three unsigned integers at DS:#100, DS:#102, and DS:#104 with the decimal values 10, 20, and 30.

```
>EU #100

3DA5:0100  130F.10
>
```

The above example replaces the integer at DS:#100 with 10.

10.3.6 Enter Words Command

■ Syntax

EW *address* [[*list*]]

The Enter Words command enters one or more word values into memory at *address*.

The optional *list* can be entered as a list of expressions separated by spaces. The expressions are entered and evaluated in the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in hexadecimal.

■ Examples

```
>EW #100 10 20 30
>
```

If the current radix is 10, the example shown above replaces the three words at DS:#100, DS:#102, and DS:#104 with the hexadecimal values A, 14, and 1E.

```
>EW #100

3DA5:0100  130F.A
>
```

The above example replaces the integer at DS:#100 with 10 (#A).

10.3.7 Enter Double Words Command

■ Syntax

ED *address* [[*list*]]

The Enter Double Words command enters one or more double-word values into memory at *address*. Double words are displayed and entered in the *segment:offset* address format; that is, two words separated by a colon (:). If the colon is omitted and only one word entered, only the offset portion of the address will be changed.

The optional *list* can be entered as a list of expressions separated by spaces. The expressions are entered and evaluated in the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in hexadecimal.

■ Examples

```
>ED #100 8700:12008
>
```

If the current radix is 10, the above example replaces the double words at DS:#100 with the hexadecimal address 21FC:2EE8 (8700:12008).

```
>ED #100

3DA5:00C8  21FC:2EE8.2EE9
>
```

The example shown above replaces the offset portion of the double word at DS:#100 with #2EE9. Since the segment portion of the address is not provided, the existing segment (#21FC) is unchanged.

10.3.8 Enter Short Reals Command

■ Syntax

ES *address* [[*list*]]

The Enter Short Reals command enters one or more short-real values into memory at *address*.

The optional *list* can be entered as a list of real numbers separated by spaces. The numbers must be entered in decimal, regardless of the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in decimal. Short-real numbers can be entered either in floating-point format or in scientific-notation format.

■ Examples

```
>ES #100 23.479 1/4 -1.65E+4 235
>
```

The above example replaces the four numbers at DS:#100, DS:#104, DS:#108, and DS:#10C with the real numbers 23.479, 0.25, -1650.0, and 235.0.

```
>ES PI
3DA5:0064   42 79 74 65    7.215589E+022   3.141593
>
```

The above example replaces the number at the symbolic address PI with 3.141593.

10.3.9 Enter Long Reals Command

■ Syntax

EL *address* [[*list*]]

The Enter Long Reals command enters one or more long-real values into memory at *address*.

The optional *list* can be entered as a list of real numbers separated by spaces. The numbers must be entered in decimal, regardless of the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in decimal. Long-real numbers can be entered either in floating-point format or in scientific-notation format.

■ Examples

```
>EL #100 23.479 1/4 -1.65E+4 235
>
```

The above example replaces the four numbers at DS:#100, DS:#108, DS:#110, and DS:#118 with the real numbers 23.479, 0.25, -1650.0, and 235.0

```
>EL PI
3DA5:0064  42 79 74 65 DC 0F 49 40    5.012391E+001  3.141593
>
```

The above example replaces the number at the symbolic address PI with 3.141593.

10.3.10 Enter 10-Byte Reals Command

■ Syntax

ET *address* [[*list*]]

The Enter 10-Byte Reals command enters one or more 10-byte-real values into memory at *address*.

The optional *list* can be entered as a list of real numbers separated by spaces. The numbers must be entered in decimal, regardless of the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in decimal. The numbers can be entered either in floating-point format or in scientific-notation format.

■ Examples

```
>ET #100 23.479 1/4 -1.65E+4 235
>
```

The above example replaces the four numbers at DS:#100, DS:#10A, DS:#114, and DS:#118 with the real numbers 23.479, 0.25, -1650.0, and 235.0.

```
>ET PI
3DA5:0064  42 79 74 65 DC 0F 49 40 7F BD  -3.292601E-193  3.141593
>
```

The above example replaces the number at the symbolic address PI with 3.141593.

10.4 Register Command

The Register command has two functions: it displays the contents of the central processing unit (CPU) registers, and it can also change the values of those registers. The modification features of the command are explained in this section. The display features of the Register command are explained in Chapter 6, "Examining Data and Expressions."

■ Mouse

The only register that can be changed with the mouse is the flags register. The register's individual bits (called flags) can be set or cleared. To change a flag, first make sure the register window is open. The window can be opened by selecting Registers from the Options menu, or by pressing the F2 key.

The flag values are shown as mnemonics in the bottom of the window. Point to the flag you want to change and click either button. The mnemonic word representing the flag value will change. The mnemonics for each flag are shown in the third and fifth columns of Table 10.1. The color or highlighting of the flag will also be reversed when you change a flag. Set flags are shown in red on color monitors and in high-intensity text on two-color monitors. Cleared flags are shown in light blue or normal text.

■ Keyboard

The registers cannot be changed with keyboard commands.

■ Dialog

To change the value of a register with a dialog command, enter a command line with the following syntax:

R [[*registername*[[=]]*expression*]]

To modify the value in a register, type the command letter **R** followed by *registername*. The CodeView debugger displays the current value of the register and prompts for a new value. Press the ENTER key if you only want to examine the value. If you want to change it, type an expression for the new value and press the ENTER key.

As an alternative, you can type both *registername* and *expression* in the same command. You can use the equal sign (=) between *registername* and *expression*, but a space has the same effect.

The register name can be any of the following names: **AX**, **BX**, **CX**, **DX**, **CS**, **DS**, **SS**, **ES**, **SP**, **BP**, **SI**, **DI**, **IP**, or **F** (for flags).

To change a flag value, supply the register name **F** when you enter the Register command. The command displays the current value of each flag as a two-letter name. The flag values are shown in Table 10.1.

Table 10.1

Flag-Value Mnemonics

Flag Name	Set Dialog	Set Window	Clear Dialog	Clear Window
Overflow	**OV**	overflow	**NV**	novrflow
Direction	**DN**	down	**UP**	up
Interrupt	**EI**	enable	**DI**	disable
Sign	**NG**	negative	**PL**	positive
Zero	**ZR**	zero	**NZ**	not zero
Auxiliary carry	**AC**	auxcarry	**NA**	no auxcy
Parity	**PE**	even	**PO**	odd
Carry	**CY**	carry	**NC**	no carry

At the end of the list of values, the command displays a dash (–). Enter new values after the dash for the flags you wish to change, then press the ENTER key. You can enter flag values in any order. Flags for which new values are not entered remain unchanged. If you do not want to change any flags, simply press the ENTER key.

If you enter an illegal flag name, an error message will be displayed. The flags preceding the error are changed; flags at and following the error are not changed.

■ Examples

```
>R IP #100
>
```

The above example changes the IP register to the value 256 (#100).

```
>R AX
AX 0E00
:_
```

The above example displays the current value of the AX register and prompts for a new value (the underscore represents the CodeView cursor).

```
>R AX
AX 0E00
:256
>_
```

You can now type any 16-bit value after the colon. As the above example shows, if the current radix is 10, you can enter 256 to change the AX value to 256 (#100).

```
>R F UP EI PL
```

The example above shows the command-line method of changing flag values.

```
>R F
NV(OV) UP(DN) EI(DI) PL(NG) NZ(ZR) AC(NA) PE(PO) NC(CY) -OV DI ZR
>R F
OV(NV) UP(DN) DI(EI) PL(NG) ZR(NZ) AC(NA) PE(PO) NC(CY) -
>
```

With the prompting method of changing flag values (shown above), the first mnemonic for each flag is the current value, and the second mnemonic (in parentheses) is the alternate value. You can enter one or more mnemonics at the dash prompt. In the example, the command is given a second time to show the results of the first command.

Chapter 11
Using System-Control Commands

11.1 Introduction 195
11.2 Help Command 195
11.3 Quit Command 196
11.4 Radix Command 197
11.5 Redraw Command 199
11.6 Screen Exchange Command 200
11.7 Search Command 201
11.8 Shell Escape Command 203
11.9 Tab Set Command 206
11.10 Redirection Commands 207
11.10.1 Redirecting CodeView Input 208
11.10.2 Redirecting CodeView Output 209
11.10.3 Redirecting CodeView Input and Output 210
11.10.4 Commands Used with Redirection 211
11.10.4.1 Comment Command 211
11.10.4.2 Delay Command 212
11.10.4.3 Pause Command 213

11.1 Introduction

This chapter discusses commands that control the operation of the CodeView debugger. The commands in this category are listed below:

Command	Action
Help (**H**)	Displays help
Quit (**Q**)	Returns to DOS
Radix (**N**)	Changes radix
Redraw (**@**)	Redraws screen
Screen Exchange (****)	Switches to output screen
Search (**/**)	Searches for regular expression
Shell Escape (**!**)	Starts new DOS shell
Tab Set (**#**)	Sets tab size
Redirection and related commands	Control redirection of CodeView output or input

The system-control commands are discussed in the following sections.

11.2 Help Command

The CodeView debugger has two help systems: a complete on-line-help system available only in window mode, and a syntax summary available with sequential mode.

■ Mouse

To enter the complete on-line-help system with the mouse, point to View on the menu bar, press a mouse button and drag the highlight down to the Help selection, then release the button. The initial help screen appears. See Chapter 3, "The CodeView Display," for details on using the on-line-help system.

■ Keyboard

If you are in window mode, press the F1 key to enter the complete on-line-help system. See Chapter 3, "The CodeView Display," for details on using the on-line-help system. If you are in sequential mode, a syntax-summary screen appears when you press F1.

■ Dialog

If you are in window mode, you can view the complete on-line-help system with the following command:

H

If you are in sequential mode, this command displays a screen containing all CodeView dialog commands with the syntax for each. This screen is the only help available in sequential mode.

11.3 Quit Command

The Quit command terminates the CodeView debugger and returns control to DOS.

■ Mouse

To quit the CodeView debugger with the mouse, point to File on the menu, press a mouse button and drag the highlight down to the Quit selection, then release the button. The CodeView screen will be replaced by the DOS screen, with the cursor at the DOS prompt.

■ Keyboard

To quit the CodeView debugger with a keyboard command, press ALT-F to open the File menu, then press ALT-Q to select Quit. The CodeView screen will be replaced by the DOS screen, with the cursor at the DOS prompt.

- **Dialog**

To quit the CodeView debugger using a dialog command, enter a command line with the following syntax:

Q

When the command is entered, the CodeView screen will be replaced by the DOS screen, with the cursor at the DOS prompt.

11.4 Radix Command

The Radix command changes the current radix for entering arguments and displaying the value of expressions. The default radix when you start the CodeView debugger is 10 (decimal). Radixes 8 (octal) and 16 (hexadecimal) can also be set. Binary and other radixes are not allowed.

The following seven conditions are exceptions; they are not affected by the Radix command:

1. The radix for entering a new radix is always decimal.
2. Format specifiers given with the Display Expression command or any of the Watch Statement commands override the current radix.
3. Addresses output by the Assemble, Dump, Enter, Examine Symbol, and Unassemble commands are always shown in hexadecimal.
4. In assembly mode, all values are shown in hexadecimal.
5. The display radix for Dump, Watch Memory, and Tracepoint Memory commands is always hexadecimal if the size is bytes, words, or double words, and always decimal if the size is integers, unsigned integers, short reals, long reals, or 10-byte reals.
6. The input radix for the Enter commands with the prompting method is always hexadecimal if the size is bytes, words, or double words, and always decimal if the size is integers, unsigned integers, short reals, long reals, or 10-byte reals. The current radix is used for all values given as part of a list, except real numbers, which must be entered in decimal.
7. The register display is always in hexadecimal.

■ Mouse

You cannot change the input radix with the mouse.

■ Keyboard

You cannot change the input radix using a keyboard command.

■ Dialog

To change the input radix using a dialog command, enter a command line with the following syntax:

N[[*radixnumber*]]

The *radixnumber* can be 8 (octal), 10 (decimal), or 16 (hexadecimal). The default radix when you start the CodeView debugger is 10 (decimal). If you give the Radix command with no argument, the debugger displays the current radix.

■ Examples

```
>N8
>? PRIME
8#153
>N10
>? PRIME
107
>N16
>? PRIME
#006b
>
```

In the above example, the value of a variable is displayed in each radix.

```
>N8
>? 34,i
28
>N10
>? 28,i
28
>N16
>? 1C,i
28
>
```

In the above example, the same number is entered in different radixes, but the `i` format specifier is used to display the result as a decimal integer in all three cases. See Chapter 6, "Examining Data and Expressions," for more information on format specifiers.

11.5 Redraw Command

The Redraw command can be used only in window mode. It redraws the CodeView screen. This command is seldom necessary, but you might need it if the output of the program being debugged temporarily disturbs the CodeView display.

■ Mouse

You cannot redraw the screen using the mouse.

■ Keyboard

You cannot redraw the screen using a keyboard command.

■ Dialog

To redraw the screen using a dialog command, enter a command line with the following syntax:

@

11.6 Screen Exchange Command

The Screen Exchange command allows you to temporarily switch from the debugging screen to the output screen.

The debugger will use either screen flipping or screen swapping to store the output and debugging screens. See Chapter 2, "Getting Started," for an explanation of flipping and swapping.

■ Mouse

To switch to the output screen with the mouse, point to View on the menu bar, press a mouse button and drag the highlight down to the Output selection, then release the button. The output screen appears. Press any key or a mouse button when you are ready to return to the debugging screen.

■ Keyboard

To switch to the output screen with a keyboard command, press the F4 key. The output screen appears. Press any key when you are ready to return to the debugging screen. This command works in either window or sequential mode.

■ Dialog

To execute the Screen Exchange command from the dialog window, enter a command line with the following syntax:

```
\
```

The output screen appears. Press any key when you are ready to return to the debugging screen.

11.7 Search Command

The Search command allows you to search for a regular expression in a source file. The expression to be found is specified either in a dialog box or as an argument to a dialog command. Once you have found an expression, you can also search for the next or previous occurrence of the expression.

Regular expressions are a method of specifying variable text patterns. A pattern can be used to search for text strings that match the pattern. This method is similar to the DOS method of using wild-card characters in file names. Regular expressions are explained in detail in Appendix B.

You can use the Search command without understanding regular expressions. Since text strings are the simplest form of regular expressions, you can simply enter a string of characters as the expression you want to find. For example, you could enter `COUNT` if you wanted to search for the word "COUNT" in the source file.

The following characters have special meanings in regular expressions: backslash (`\`), asterisk (`*`), left bracket ([), period (.), dollar sign ($), and caret (^). To find strings containing these characters, you must precede the characters with a backslash; this cancels their special meanings.

For example, you would use `\*` to find `x*y`. The periods in the relational operators must also be preceded by a backslash.

The Case Sense selection from the Options menu has no effect on searches for regular expressions.

Important

When you search for the next occurrence of a regular expression, the CodeView debugger searches to the end of the file, then wraps around and begins again at the start of the file. This can have unexpected results if the expression occurs only once. When you give the command repeatedly, nothing seems to happen. Actually, the debugger is repeatedly wrapping around and finding the same expression each time.

■ Mouse

To find a regular expression with the mouse, point to Search on the menu bar, press a mouse button and drag the highlight down to the Find selection, then release the button. A dialog box appears, asking for the regular expression to be found. Type the expression, and press either the ENTER key or a mouse button. The CodeView debugger starts searching at the current cursor position and puts the cursor at the next line containing the regular expression. An error message appears if the expression is not found. If you are in assembly mode, the debugger automatically switches to source mode when the expression is found.

After you have found a regular expression, you can search for the next or previous occurrence of the expression. Point to Search on the menu bar, press a mouse button and drag the highlight down to the Next or Previous selection, then release the button. The cursor will move to the next or previous match of the expression.

You can also search the executable code for a label (such as a routine name or an assembly-language label). Point to Search on the menu bar, press a mouse button and drag the highlight down to the Label selection, then release the button. A dialog box appears, asking for the label to be found. Type the label name, and press either the ENTER key or a mouse button. The cursor will move to the line containing the label. This selection differs from other search selections because it searches executable code rather than source code. The CodeView debugger will switch to assembly mode, if necessary, to display a label in a library routine or assembly-language module.

■ Keyboard

To find a regular expression with a keyboard command, press ALT-S to open the Search menu, then press ALT-F to select Find. A dialog box appears, asking for the regular expression to be found. Type the expression, and press the ENTER key. The CodeView debugger starts searching at the current cursor position and puts the cursor at the next line containing the regular expression. An error message appears if the expression is not found. If you are in assembly mode, the debugger automatically switches to source mode when the expression is found.

After you have found a regular expression, you can search for the next or previous occurrence of the expression. Press ALT-S to open the Search menu, then press ALT-N to select Next or ALT-P to select Previous. The cursor will move to the next or previous match of the expression.

You can also search the executable code for a label (such as a routine name or an assembly-language label). Press ALT-S to open the Search menu, then press ALT-L to select Label. A dialog box appears, asking for the label to be found. Type the label name, and press the ENTER key. The cursor will move to the line containing the label. This selection differs from other search selections because it searches executable code rather than source code. The CodeView debugger will switch to assembly mode, if necessary, to display a label in a library routine or assembly-language module.

■ Dialog

To find a regular expression using a dialog command, enter a command line with the following syntax:

/[[*regularexpression*]]

If *regularexpression* is given, the CodeView debugger searches the source file for the first line containing the expression. If no argument is given, the debugger searches for the next occurrence of the last regular expression specified.

In window mode, the CodeView debugger starts searching at the current cursor position and puts the cursor at the next line containing the regular expression. In sequential mode, the debugger starts searching at the last source line displayed. It puts the source line where the expression is found on the screen. An error message appears if the expression is not found. If you are in assembly mode, the CodeView debugger automatically switches to source mode when the expression is found.

You cannot search for a label with the dialog version of the Search command, but using the View command with the label as an argument has the same effect.

11.8 Shell Escape Command

The Shell Escape command allows you to exit from the CodeView debugger to a DOS shell. You can execute DOS commands or programs from within the debugger, or you can exit from the debugger to DOS while retaining your current debugging context.

The Shell Escape command works by saving the current processes in memory and then executing a second copy of **COMMAND.COM**. The **COMSPEC** environment variable is used to locate a copy of **COMMAND.COM**.

Opening a shell requires a significant amount of free memory (usually more than 200K). This is because the CodeView debugger, the symbol table, **COMMAND.COM**, and the program being debugged must all be saved in memory. If you do not have enough memory, an error message will appear. Even if you have enough memory to start a new shell, you may not have enough memory left to execute large programs from the shell.

If you change directories while working in the shell, make sure you return to the original directory before returning to the CodeView debugger. If you don't, the debugger may not be able to find and load source files when it needs them.

Note

In order to use the Shell Escape command, the executable file being debugged must release the memory it does not need. Programs compiled with the Microsoft FORTRAN Optimizing Compiler do this automatically if the FORTRAN start-up code has been executed. You must execute into the program before using the Shell Escape command; for example, enter `G main` after starting the CodeView debugger.

You cannot use the Shell Escape command with assembler programs unless the program specifically releases memory using the DOS function call #4A (Set Block). The same thing can be accomplished by linking the assembler program with the **/CPARMAXALLOC** link option. If the program has not released memory, the CodeView debugger will print this message: `Not enough memory`.

■ Mouse

To open a DOS shell with the mouse, point to File on the menu bar, press a mouse button and drag the highlight down to the Shell selection, then release the button. If there is enough memory to open the shell, the DOS screen will appear. You can execute any DOS internal command or any program. When you are ready to return to the debugging session, type the command `exit` (in any combination of uppercase and lowercase letters). The debugging screen will appear with the same status it had when you left it.

■ Keyboard

To open a DOS shell using a keyboard command, press ALT-F to open the File menu, then press ALT-S to select Shell. If there is enough memory to open the shell, the DOS screen will appear. You can execute any DOS internal command or any program. When you are ready to return to the debugging session, type the command `exit` (in any combination of uppercase and lowercase letters). The debugging screen will appear with the same status it had when you left it.

■ Dialog

To open a DOS shell using a dialog command, enter a command line with the following syntax:

!⟦*command*⟧

If you want to exit to DOS and execute several programs or commands, enter the command with no arguments. The CodeView debugger executes a new copy of **COMMAND.COM** and the DOS screen appears. You can run programs or DOS internal commands. When you are ready to return to the debugger, type the command `exit` (in any combination of uppercase and lowercase letters). The debugging screen will appear with the same status it had when you left it.

If you want to execute a program or DOS internal command from within the CodeView debugger, enter the Shell Escape command (!) followed by the name of the command or program you want to execute. The output screen appears and the debugger executes the command or program. When the output from the command or program is finished, the message `Press any key to continue...` appears at the bottom of the screen. Press a key to make the debugging screen reappear with the same status it had when you left it.

■ Examples

```
>!
```

In the above example, the CodeView debugger saves the current debugging context and executes a copy of **COMMAND.COM**. The DOS screen appears and you can enter any number of commands. To return to the debugger, enter `exit`.

```
>!DIR a:*.for
```

In the above example, the DOS internal command `DIR` is executed with the argument `a:*.for`. The directory listing will be followed by a prompt telling you to press any key to return to the CodeView debugging screen.

```
>!CHKDSK a:
```

In the above example, the DOS external command `CHKDSK` is executed, and the status of the disk in Drive A is displayed in the dialog window. The program name specified could be for any executable file, not just for a DOS external program.

11.9 Tab Set Command

The Tab Set command sets the width in spaces that the CodeView debugger fills for each tab character. The default tab is eight spaces. You might want to set a smaller tab size if your source code has so many levels of indentation that source lines extend beyond the edge of the screen. This command has no effect if your source code was written with an editor that indents with spaces rather than with tab characters.

■ Mouse

You cannot set the tab size using the mouse.

■ Keyboard

You cannot set the tab size using a keyboard command.

■ Dialog

To set the tab size using a dialog command, enter a command line with the following syntax:

#number

The *number* is the new number of characters for each tab character. In window mode, the screen will be redrawn with the new tab width when you enter the command. In sequential mode, any output of source lines will reflect the new tab size.

■ Example

```
>.
32:                    IF (X(I)) .LE. X(J)) GOTO 301
>#4
>.
32:          IF (X(I)) .LE. X(J)) GOTO 301
>
```

In the above example, the Source Line (.) command is used to show the source line with the default tab width of eight spaces. Next the Tab Set command is used to set the tab width to four spaces. The Source Line command then shows the same line.

11.10 Redirection Commands

The CodeView debugger provides several options for redirecting commands from or to devices or files. In addition to the redirection commands, several other commands are relevant only when used with redirected files. The redirection commands and related commands are discussed in Sections 11.10.1 – 11.10.4.3.

■ Mouse

None of the redirection or related commands can be executed with the mouse.

■ Keyboard

None of the redirection or related commands can be executed with keyboard commands.

■ Dialog

The redirection commands are entered with dialog commands, as shown in Sections 11.10.1 – 11.10.4.3.

11.10.1 Redirecting CodeView Input

■ Syntax

< *devicename*

The Redirected Input command causes the CodeView debugger to read all subsequent command input from a device, such as another terminal or a file. The sample session supplied with most versions of the debugger is an example of commands being redirected from a file.

■ Examples

```
><COM1
```

The above example redirects commands from the device (probably a remote terminal) designated as `COM1` to the CodeView terminal.

```
><INFILE.TXT
```

The above example redirects command input from file `INFILE.TXT` to the CodeView debugger. You might use this command to prepare a CodeView session for someone else to run. You create a text file containing a series of CodeView commands separated by carriage-return–line-feed combinations or semicolons. When you redirect the file, the debugger will execute the commands to the end of the file. One way to create such a file is to redirect commands from the CodeView debugger to a file (see Section 11.10.3) and then edit the file to eliminate the output and add comments.

11.10.2 Redirecting CodeView Output

■ Syntax

[[T]>[>]] *devicename*

The Redirected Output command causes the debugger to write all subsequent command output to a device, such as another terminal, a printer, or a file. The term "output" includes not only the output from commands, but the command characters that are echoed as you type them.

The optional **T** indicates that the output should be echoed to the CodeView screen. If you do not use the **T**, you will not be able to see your commands as you type them. Normally, you will want to use the **T** if you are redirecting output to a file, so that you can see what you are typing. However, if you are redirecting output to another terminal, you may not want to see the output on the CodeView terminal.

The optional second greater-than symbol appends the output to an existing file. If you redirect output to an existing file without this symbol, the existing file will be replaced.

■ Examples

```
>>COM1
```

The above example's output is redirected to the device designated as COM1 (probably a remote terminal). One situation in which you might want to do this is when you are debugging a graphics program and want CodeView commands to be displayed on a remote terminal at the same time that the program display appears on the originating terminal.

```
>T>OUTFILE.TXT
.
.
.
>>CON
.
.
.
```

The above example's output is redirected to the file OUTFILE.TXT. You might want to do this in order to keep a permanent record of a CodeView session. Note that the optional **T** is used so that the session will be echoed to the CodeView screen as well as to the file. After redirecting some commands to a file, output is returned to the console (terminal) with the command >CON.

```
>T>>OUTFILE.TXT
```

If, later in the session, you want to redirect more commands to the same file, use the double greater-than symbol, as in the above example, to append the output to the existing file.

11.10.3 Redirecting CodeView Input and Output

■ Syntax

= *devicename*

The Redirected Input and Output command causes the CodeView debugger to write all subsequent command output to a device and to simultaneously receive input from the same device. This is practical only if the device is a remote terminal.

Redirecting input and output works best if you start in sequential mode (using the **/T** option), since this eliminates unnecessary screen exchanges. The CodeView debugger's window interface has little purpose in this situation, since the remote terminal can act only as a sequential (nonwindow) device.

■ Example

```
>=COM1
```

In the above example, output and input are redirected to the device designated as COM1. This would be useful if you wanted to enter debugging commands and see the debugger output on a remote terminal, while entering program commands and viewing program output on the terminal where the debugger is running.

11.10.4 Commands Used with Redirection

The following commands are intended for use when redirecting commands to or from a file. Although they are always available, these commands have little practical use during a normal debugging session.

Command	Action
Comment (*)	Displays comment
Delay (:)	Delays execution of commands from a redirected file
Pause (")	Interrupts execution of commands from a redirected file until a key is pressed

11.10.4.1 Comment Command

■ Syntax

**comment*

The Comment command is an asterisk (*) followed by text. The CodeView debugger echoes the text of the comment to the screen (or other output device). This command is useful in combination with the redirection commands when saving a commented session, or when writing a commented session that will be redirected to the debugger.

■ Examples

```
>T>OUTPUT.TXT
>* Dump first 20 bytes of screen buffer
>D #B800:0 L 20
B800:0000 54 17 6F 17 20 17 72 17 65 17 74 17 75 17 72 17 T.o. .r.e.t.u.r.
B800:0010 6E 17 20 17                                     n. .
>
```

In the above example, the user is sending a copy of a CodeView session to file OUTPUT.TXT. Comments are added to explain the purpose of the command. The text file will contain commands, comments, and command output.

```
* Dump first 20 bytes of screen buffer
D #B800:0 L 20
.
.
.
< CON
```

The above example illustrates another way to use the Comment command. You can put comments into a text file of commands that will be executed automatically when you redirect the file into the CodeView debugger. In this example, an editing program was used to create the text file called INPUT.TXT.

```
><INPUT.TXT
>* Dump first 20 bytes of screen buffer
>D #B800:0 L 20
B800:0000 54 17 6F 17 20 17 72 17 65 17 74 17 75 17 72 17 T.o. .r.e.t.u.r.
B800:0010 6E 17 20 17                                     n. .
.
.
.
>< CON
>
```

When you read the file into the debugger, using the Redirected Input command, you will see the comment, then the output from the command, as in the above example.

11.10.4.2 Delay Command

- **Syntax**

```
:
```

The Delay command interrupts execution of commands from a redirected file and waits about half a second before continuing. You can put multiple Delay commands on a single line to increase the length of the delay. The delay is the same length, regardless of the processing speed of the computer.

■ **Example**

```
: ;* That was a short delay...
::::: ;* That was a longer delay...
```

In the above example from a text file that might be redirected into the CodeView debugger, the Delay command is used to slow execution of the redirected file.

11.10.4.3 Pause Command

■ **Syntax**

```
"
```

The Pause command interrupts execution of commands from a redirected file and waits for the user to press a key. Execution of the redirected commands begins as soon as a key is pressed.

■ **Example**

```
* Press any key to continue
"
```

In the above example from a text file that might be redirected into the CodeView debugger, a Comment command is used to prompt the user to press a key. Then the Pause command is used to halt execution until the user responds.

```
>* Press any key to continue
>"
```

The above example shows the output when the text is redirected into the debugger. The next CodeView prompt will not appear until the user presses a key.

Appendixes

Appendixes

A Command and Mode Summary 217
B Regular Expressions 229
C Error Messages 237

Appendix A
Command and Mode Summary

A.1 Introduction 219
A.2 Modes 219
A.3 Options 221
A.4 Window Commands 222
A.5 Dialog Commands 224
A.6 Format Specifiers 227

A.1 Introduction

This appendix summarizes the CodeView debugger's modes, options, and commands.

A.2 Modes

Many CodeView commands and options deal with switching between modes. The debugger has four types of modes: screen modes, debugging modes, display modes, and exchange modes. The modes for each of these types are shown in Table A.1.

Table A.1
CodeView Modes

Type	Mode	Purpose	Command or Option
Screen	Debugging	Debugs program	Default
	Output	Views program output	F4 Screen Exchange (\) Output from View menu
	Help	Views command and option summary	F1 Help from View menu
Debugging	Window	Views program in windows	Default for IBM **/W** option
	Sequential	Views program sequentially	Default for non-IBM **/T** option
Display	Source	Displays source lines	Default start-up for FORTRAN programs F3 **S+** Source from View menu
	Assembly	Displays assembly instructions	Default start-up for programs with no symbolic information F3 **S–** Assembly from View menu
	Mixed	Displays source and assembly	**S&** Mixed from Options menu
Exchange	Flipping	Flips between video pages	Default for IBM **/F** option
	Swapping	Swaps between buffers	**/S** option

A.3 Options

The following start-up options are available with the CodeView debugger:

Option	Effect
/B	Starts in black and white with CGA
/C*commands*	Executes commands on start-up
/D	Turns off nonmaskable interrupt and 8259 interrupt trapping
/F	Starts with screen flipping (exchanges screens by flipping video pages)
/I	Forces the debugger to handle nonmaskable interrupt and 8259 interrupt trapping on non-IBM-compatible machines
/M	Disables the mouse
/T	Starts in sequential mode
/S	Starts with screen swapping (exchanges screens by changing buffers)
/W	Starts in window mode
/2	Permits use of two monitors
/43	Starts in EGA 43-row mode

A.4 Window Commands

Table A.2 shows the keyboard and mouse versions of each window command.

Table A.2
Window Commands

Action	Keyboard	Mouse
Go to help screen	F1	Help from View menu
Open register window	F2	Registers from Options menu
Toggle source/assembly	F3	Source/Assembly from View menu
Switch to output screen	F4	Output from View menu
Go	F5	Click either on Go!
Switch to display/dialog	F6	None
Execute to here	F7 at location	Click right at location
Trace through procedure	F8	Click left on Trace!
Set breakpoint here	F9 at location	Click left at location
Step over procedure	F10	Click right on Trace!
Change flag	None	Click either on flag
Move separator line up	CONTROL-U	Drag line up
Move separator line down	CONTROL-D	Drag line down
Scroll up line in window	None	Click left on up arrow
Scroll up page in window	PGUP	Click right on up arrow
Scroll to top of window	HOME	Click both on up arrow
Scroll down line in window	None	Click left on down arrow
Scroll down page in window	PGDN	Click right on down arrow
Scroll to bottom of window	END	Click both on down arrow
Move cursor	UP ARROW/ DOWN ARROW	None

Table A.3 lists and explains the selections on the CodeView menus.

Table A.3
Menu Selections

Menu	Selection	Action
File	Load...	Loads new text or source file
	Shell	Starts new DOS shell
	Quit	Quits the CodeView debugger
Search	Find...	Finds first regular expression
	Next	Finds next regular expression
	Previous	Finds previous regular expression
	Label...	Finds routine or code label
View	Help	Opens on-line-help screen
	Source	Displays source lines
	Assembly	Displays assembly
	Output	Switches to output screen
	Evaluate...	Evaluates expression
Run	Start	Restarts current program and runs
	Restart	Restarts current program
	Execute	Executes in slow motion
	Clear Breakpoints	Clears all breakpoints
Watch	Add Watch...	Sets watch expression
	Watchpoint...	Sets watchpoint
	Tracepoint...	Sets tracepoint
	Delete Watch...	Deletes watch statement
Options	Flip/Swap	Toggles screen exchange
	Mix Source	Toggles mixed source and instructions
	Symbols	Toggles symbolic reference to variables
	Bytes Coded	Toggles byte display for instructions
	Registers	Toggles register window
	Case Sense	Toggles case sensitivity of symbols
	C	Turns on C expression evaluator
	FORTRAN	Turns on FORTRAN expression evaluator
Calls	*routine*	Goes to *routine* call line

A.5 Dialog Commands

The CodeView dialog commands and the syntax for each are listed alphabetically in this section. Many of the commands require a type to indicate the size of the data accepted by the command. The types used by the Dump, Enter, Watch Memory, and Tracepoint Memory commands are listed below in order of their data size:

Type	Description
No type	The default type (the last one used with a Dump, Enter, Watch Memory, or Tracepoint Memory command, or byte if none of these commands has been used)
A (ASCII)	ASCII (8-bit) characters
B (Byte)	Byte (8-bit) hexadecimal values
I (Integer)	Integer (16-bit) decimal values; equivalent to FORTRAN **INTEGER*2** on DOS systems
U (Unsigned)	Unsigned (8-bit) decimal values
W (Word)	Word (16-bit) hexadecimal values; no FORTRAN equivalent
D (Double Word)	Double-word (32-bit) hexadecimal values; no FORTRAN equivalent
S (Short Real)	Short-real (32-bit) values; equivalent to FORTRAN **REAL*4**
L (Long Real)	Long-real (64-bit) values; equivalent to FORTRAN **REAL*8**
T (10-Byte Real)	10-byte-real values; no FORTRAN equivalent

The CodeView dialog commands and the syntax for each are shown in Table A.4.

Table A.4
Dialog Commands

Name	Syntax	Description
Assemble	**A** [[*address*]]	Assembles mnemonics starting at *address*
Breakpoint Clear	**BC** *list*	Clears breakpoints in *list*
Breakpoint Disable	**BD** *list*	Disables breakpoints in *list*
Breakpoint Enable	**BE** *list*	Enables breakpoints in *list*
Breakpoint List	**BL**	Lists breakpoints with status of each
Breakpoint Set	**BP** [[*address*[[*count*]][["*cmds*"]]]	Sets breakpoint at *address*; *count* is pass count; *cmds* are commands to be executed at each break
Comment	* *comment*	Displays explanatory text
Current Location	.	Displays the current source line
Delay	:	Delays execution of redirected commands (may be repeated for longer delays)
Display Expression	? *expression*[[,*format*]]	Displays *expression* in *format*
Dump	**D**[[*type*]] [[*address* | *range*]]	Dumps memory range in *type* format
Enter	**E**[[*type*]] *address* [[*list*]]	Enters memory value in *type* format
Examine Symbols	**X?**[[*mod*!]][[*rout*.]][[*sym*]][[*]]	Displays specified symbols
Execute	**E**	Executes in slow motion

Table A.4 *(continued)*

Name	Syntax	Description
Go	**G** [[*address*]]	Executes to *address* or to end
Help	**H**	Displays on-line help
Load	**L** [[*arguments*]]	Restarts program
Pause	**"**	Interrupts execution of redirected commands and waits for keystroke
Program Step	**P** [[*count*]]	Executes source lines or instructions, stepping over routine, procedure, and interrupt calls; repeats *count* times
Quit	**Q**	Exits and returns to DOS
Radix	**N**[[*radixnumber*]]	Sets input radix
Redirection	[[**T**]]**>**[[**>**]]*devicename* **<***devicename* **=***devicename*	Redirects input or output to or from *devicename*
Redraw	**@**	Redraws the screen
Register	**R**[[*register* [[**=**]]*expression*]]]	Displays registers and flags, or sets new registers and flags
Screen Exchange	****	Exchanges the CodeView and output screens
Search	**/**[[*regularexpression*]]	Searches for a regular expression
Set Mode	**S**[[**+** | **−** | **&**]]	Sets display mode to source, assembly, or mixed
Shell Escape	**!**[[*command*]]	Escapes to a new DOS shell
Stack Trace	**K**	Displays active routines on the stack
Tab Set	**#***number*	Sets *number* of spaces for each tab character

Table A.4 *(continued)*

Name	Syntax	Description
Trace	**T** [[*count*]]	Executes source lines or instructions, tracing into routine, procedure, or interrupt calls; repeats *count* times
Tracepoint	**TP?** *expression*[[,*format*]] **TP**[[*type*]] {*address* | *range*}	Breaks when the memory location specified by *expression* or *range* changes; displays in watch window
Unassemble	**U** [[*address* | *range*]]	Displays unassembled instructions
Use	**USE**[[*language*]]	Switches expression evaluators
View	**V** [[*expression*]]	Displays source lines
Watch	**W?** *expression*[[,*format*]] **W**[[*type*]] {*address* | *range*}	Displays *expression* or *range* in watch window
Watch Delete	**Y** *number*	Deletes (yanks) watch statements
Watch List	**W**	Lists tracepoints and watchpoints
Watchpoint	**WP?** *expression*[[,*format*]]	Breaks when *expression* is true; displays in watch window
8087	**7**	Displays 8087 registers

A.6 Format Specifiers

Several commands allow you to specify the format in which expression values are displayed. The following commands can have formatted output:

? *expression*[[,*format*]]	Expression command
W? *expression*[[,*format*]]	Watch command
WP? *expression*[[,*format*]]	Watchpoint command
TP? *expression*[[,*format*]]	Tracepoint command

The *format* in these commands can be a CodeView format specifier from among those listed in Table A.5.

Table A.5
Format Specifiers

Character	Argument Type	Output Format
d	Integer	Signed decimal integer
i	Integer	Signed decimal integer
u	Integer	Unsigned decimal integer
o	Integer	Unsigned octal integer
x or **X**	Integer	Hexadecimal integer
f	Floating point	Signed value in floating-point decimal format with six decimal places
e or **E**	Floating point	Signed value in scientific-notation format with up to six decimal places (trailing zeros or decimal point truncated)
g or **G**	Floating point	Signed value with floating-point decimal or scientific notation, whichever is more compact
c	Character	Single character
s	String	Characters printed up to the first null character

The prefix **h** can be used with the integer format specifiers (**d**, **o**, **u**, **x**, and **X**) to specify an **INTEGER*2**. The prefix **l** can be used with the same types to specify an **INTEGER*4**.

Appendix B
Regular Expressions

B.1 Introduction 231
B.2 Special Characters in Regular Expressions 231
B.3 Searching for Special Characters 232
B.4 Using the Period 232
B.5 Using Brackets 233
B.5.1 Using the Dash within Brackets 233
B.5.2 Using the Caret within Brackets 234
B.5.3 Matching Brackets within Brackets 234
B.6 Using the Asterisk 234
B.7 Matching the Start or End of a Line 235

B.1 Introduction

Regular expressions are used to specify text patterns in searches for variable text strings. Special characters can be used within regular expressions to specify groups of characters to be searched for.

This appendix explains all the special characters you can use to form regular expressions, but you do not need to learn them all to use the CodeView Search commands. The simplest form of regular expression is simply a text string. For example, if you want to search for all instances of the symbol COUNT, you can specify COUNT as the string to be found.

If you only want to search for simple strings, you do not need to read this entire appendix, but you should know how to search for strings containing the special characters used in regular expressions. See Section B.3 for more information.

B.2 Special Characters in Regular Expressions

The following characters have special meanings in regular expressions:

Character	Purpose
Backslash (\)	Removes the special characteristics of the following characters: backslash (\), period (.), caret (^), dollar sign ($), asterisk (*), and left bracket ([)
Period (.)	Matches any character
Caret (^)	Matches beginning of line
Dollar sign ($)	Matches end of line
Asterisk (*)	Matches any number of repetitions of the previous character
Brackets ([ ])	Matches characters specified within the brackets; the following special characters may be used inside brackets:

Caret (^)	Reverses the function of the brackets; that is, matches any character except those specified within the brackets
Dash (-)	Matches characters in ASCII order between (inclusive) the characters on either side of the dash

B.3 Searching for Special Characters

If you need to match one of the special characters used in regular expressions, you must precede it with a backslash when you specify a search string. The special characters are the asterisk (*), backslash (\), left bracket ([), caret (^), dollar sign ($), and period (.).

For example, the regular expression `I*J` matches such combinations as `IJ`, `I+J`, and `I-J`. The regular expression `I\*J` matches only `I*J`. The backslash is necessary because the asterisk (*) is a special character in regular expressions.

B.4 Using the Period

A period in a regular expression matches any single character. This corresponds to the question mark (?) used in specifying DOS file names.

For example, you could use the regular expression `AMAX.` to search for either of the intrinsic functions `AMAX0` and `AMAX1`. You could use the expression `X.Y` to search for strings such as `X+Y`, `X-Y`, or `X*Y`. If your programming style is to put a space between variables and operators, you could use the regular expression `X . Y` for the same purpose.

Note that when you use the period as a wild card, you will find the strings you are looking for, but you may also find other strings that you aren't interested in. You can use brackets to be more exact about the strings you want to find.

B.5 Using Brackets

You can use brackets to specify a character or characters you want to match. Any of the characters listed within the brackets is an acceptable match. This method is more precise than using a period to match any character.

For example, the regular expression `x[-+/*]y` matches `x+y`, `x-y`, `x/y`, or `x*y`, but not `x=y` or `xzy`. The regular expression `COUNT[12]` matches `COUNT1` and `COUNT2`, but not `COUNT3`.

Most regular-expression special characters have no special meaning when used within brackets. The only special characters within brackets are the dash (−), caret (^), and right bracket (]). Even these characters only have special meanings in certain contexts, as explained in Sections B.5.1 – B.5.3.

B.5.1 Using the Dash within Brackets

The dash can be used within brackets to specify a group of sequential ASCII characters. For example, the regular expression `[0-9]` matches any digit; it is equivalent to `[0123456789]`. Similarly, `[a-z]` matches any lowercase letter, and `[A-Z]` matches any uppercase letter.

You can combine ASCII ranges of characters with other listed characters. For example, `[A-Za-z ]` matches any uppercase or lowercase letter or a space.

The dash has this special meaning only if you use it to separate two ASCII characters. It has no special meaning if used directly after the starting bracket or directly before the ending bracket. This means that you must be careful where you place the dash (minus sign) within brackets.

For example, you might use the regular expression `[+-/*]` to match the characters `+`, `-`, `/`, and `*`. However, this does not give the intended result. Instead it matches the characters between `+` and `/` and also the character `*`. To specify the intended characters, put the dash first or last in the list: `[-+/*]` or `[+/*-]`.

B.5.2 Using the Caret within Brackets

If used as the first character within brackets, the caret (^) reverses the meaning of the brackets. That is, any character except the ones in brackets will be matched. For example, the regular expression `[^0-9]` matches any character that is not a digit. Specifying the characters to be excluded is often more concise than specifying the characters you want to match.

If the caret is not in the first position within the brackets, it is treated as an ordinary character. For example, the expression `[0-9^]` matches any digit or a caret.

B.5.3 Matching Brackets within Brackets

Sometimes you may want to specify the bracket characters as characters to be matched. This is no problem with the left bracket; it is treated as a normal character. However, the right bracket is interpreted as the end of the character list rather than as a character to be matched.

If you want the right bracket to be matched, you must make it the first character after the initial left bracket. For example, the regular expression `[]#![@%]` matches either bracket character or any of the other characters listed within the brackets. However, if you changed the order of just one of the characters (to `[#]![@%]`), the meaning would be changed so that you would be specifying two groups of characters in brackets: `[#]` and `[@%]`.

B.6 Using the Asterisk

The asterisk matches zero or more occurrences of the character preceding the asterisk.

For example, the regular expression `IF *(TEST` will match any of the following strings:

```
IF (TEST
IF        (TEST
IF(TEST
```

Note that the last example contains zero repetitions of the space character.

The asterisk is convenient if the text you are searching for might contain some spaces, but you don't know the exact number. (Be careful in this situation: you can't be sure if the text contains a series of spaces or a tab.)

You might also use the asterisk to search for a symbol when you aren't sure of the spelling. For example, you could use `first*ime` if you aren't sure if the identifier you are searching for is spelled `firsttime` or `firstime`.

One particularly powerful use of the asterisk is to combine it with the period (`.*`). This combination searches for any group of characters, and is similar to the asterisk used in specifying DOS file names. For example, the expression `(.*)` matches `(test)`, `(response .EQ. 'Y')`, `(x=0;x .LE. 20;x=x+1)`, or any other string that starts with a left parenthesis and ends with a right parenthesis.

You can use brackets with the asterisk to search for a sequence of repeated characters of a given type. For example, `\[[0-9]*]` matches number strings within brackets (`[1353]` or `[3]`), but does not match character strings within brackets (`[count]`). Empty brackets (`[ ]`) are also matched, since the characters in the brackets are repeated zero times.

B.7 Matching the Start or End of a Line

In regular expressions, the caret (^) matches the start of a line, while the dollar sign ($) matches the end of a line.

For example, the regular expression `^C` matches any uppercase C that starts a line. Similarly, `)$` matches a right parenthesis at the end of a line, but not a right parenthesis within a line.

You can combine both symbols to search for entire lines. For example, `^{$` matches any line consisting of only a left curly brace in the left margin, and `^$` matches blank lines.

Error Messages

Appendix C
Error Messages

The CodeView debugger displays an error message whenever it detects a command it cannot execute. Most errors (start-up errors are the exception) terminate the CodeView command under which the error occurred, but do not terminate the debugger. You may see any of the following error messages:

`Argument to IMAG/DIMAG must be simple type`

You specified an argument to an **IMAG** or **DIMAG** function that is not permitted, such as an array with no subscripts.

`Array must have subscript`

You specified an array without any subscripts in an expression: for example, `IARRAY+2`.

`Bad address`

You specified the address in an invalid form.

For instance, you may have entered an address containing hexadecimal characters when the radix is decimal.

`Bad breakpoint command`

You typed an invalid breakpoint number with the Breakpoint Clear, Breakpoint Disable, or Breakpoint Enable command.

The number must be in the range 0 to 19.

`Bad flag`

You specified an invalid flag mnemonic with the Register dialog command (**R**).

Use one of the mnemonics displayed when you enter the command **RF**.

`Bad format string`

You used an invalid format specifier following an expression.

Expressions used with the Display Expression, Watch, Watchpoint, and Tracepoint commands can have CodeView format specifiers set off from the expression by a comma. The valid format specifiers are **d**, **i**, **u**, **o**, **x**, **X**, **f**, **e**, **E**, **g**, **G**, **c**, and **s**. Some format specifiers can be preceded by the prefix **h** or **l**. See Chapter 6, "Examining Data and Expressions," for more information about format specifiers.

`Bad integer or real constant`

You specified an illegal numeric constant in an expression.

`Bad intrinsic function`

You specified an illegal intrinsic function name in an expression.

`Bad radix (use 8, 10, or 16)`

With the **N** command you can use only octal, decimal, and hexadecimal radixes.

`Bad register`

You typed the Register command (**R**) with an invalid register name.

Use **AX**, **BX**, **CX**, **DX**, **SP**, **BP**, **SI**, **DI**, **DS**, **ES**, **SS**, **CS**, **IP**, or **F**.

`Bad subscript`

You entered an illegal subscript expression for an array, such as `IARRAY(3.3)` or `IARRAY((3,3))`.

`Bad type cast`

The types of the operands in an expression are incompatible.

`Bad type (use one of 'ABDILSTUW')`

The valid dump types are ASCII (**A**), Byte (**B**), Integer (**I**), Unsigned (**U**), Word (**W**), Double Word (**D**), Short Real (**S**), Long Real (**L**), and 10-Byte Real (**T**).

`Badly formed type`

The type information in the symbol table of the file you are debugging is incorrect.

If this message occurs, please note the circumstances of the error and inform Microsoft Corporation, using the Software Problem Report at the back of this manual.

`Breakpoint # or '*' expected`

You entered the Breakpoint Clear (**BC**), Breakpoint Disable (**BD**), or Breakpoint Enable (**BE**) command with no argument.

These commands require that you specify the number of the breakpoint to be acted on, or that you specify the asterisk (*), indicating that all breakpoints are to be acted on.

`Cannot use struct or union as scalar`

A struct or union variable cannot be used as a scalar value in a C expression.

Such variables must be followed by a file specifier or preceded with the address-of operator.

`Cannot cast complex constant component into REAL`

Both the real and imaginary components of a **COMPLEX** constant must be compatible with type **REAL**.

`Cannot cast IMAG/DIMAG argument to COMPLEX`

Arguments to **IMAG** and **DIMAG** must be simple numeric types.

`Can't find` *filename*

The CodeView debugger could not find the executable file you specified when you started.

You probably misspelled the file name, or the file is located in a different directory.

`Character constant too long`

You specified a character constant that is too long for the FORTRAN expression evaluator.

The limit is 126 bytes.

`Character too big for current radix`

In a constant, you specified a radix that is larger than the current CodeView radix.

Use the **N** command to change the radix.

Constant too big

The CodeView debugger cannot accept an unsigned constant number larger than 4,294,967,295 (16#FFFFFFFF).

Divide by zero

An expression in an argument of a dialog command attempts to divide by zero.

Expression too complex

An expression given as a dialog-command argument is too complex.

Try simplifying the expression.

Extra input ignored

You specified too many arguments to a command.

The CodeView debugger evaluates the valid arguments and ignores the rest. Often, in this situation, the debugger may not evaluate the arguments the way you intended.

Flip/Swap option off – application output lost

The program you are debugging is writing to the screen, but the output cannot be displayed because you have turned off the flip/swap option.

Floating point error

This message should not occur, but if it does, please note the circumstances of the error and inform Microsoft Corporation, using the Software Problem Report at the back of this manual.

Index out of bound

You specified a subscript value that is outside the bounds declared for the array.

Internal debugger error

If this message occurs, please note the circumstances of the error and inform Microsoft Corporation, using the Software Problem Report at the back of this manual.

Invalid argument

One of the arguments you specified is not a valid CodeView expression.

`Missing '"'`

You specified a string as an argument to a dialog command, but you did not supply a closing double quotation mark.

`Missing '('`

An argument to a dialog command was specified as an expression containing a right parenthesis, but no left parenthesis.

`Missing ')'`

An argument to a dialog command was specified as an expression containing a left parenthesis, but no right parenthesis.

`Missing ']'`

An argument to a dialog command was specified as an expression containing a left bracket, but no right bracket.

This error can also occur if a regular expression is specified with a right bracket but no left bracket.

`Missing '(' in complex constant`

The debugger is expecting the opening parenthesis of a complex constant in an expression, but it is missing.

`Missing ')' in complex constant`

The debugger expects the closing parenthesis of a complex constant in an expression.

`Missing ')' in substring`

The debugger expects the closing parenthesis of a substring expression.

`Missing '(' to intrinsic`

The debugger expects an opening parenthesis for an intrinsic function.

`Missing ')' to intrinsic`

The debugger expects a closing parenthesis for an intrinsic function.

`No closing single quote`

You specified a character in an expression used as a dialog-command argument, but the closing single quotation mark is missing.

No code at this line number

You tried to set a breakpoint on a source line that does not correspond to code.

For instance, the line may be a data declaration or a comment.

No match of regular expression

No match was found for the regular expression you specified with the Search command or with the Find selection from the Search menu.

No previous regular expression

You selected Previous from the Search menu, but there was no previous match for the last regular expression specified.

No program to debug

You have executed to the end of the program you are debugging.

You must restart the program (using the Restart command) before using any command that executes code.

No source lines at this address

The address you specified as an argument for the View command (**V**) does not have any source lines.

For instance, it could be an address in a library routine or an assembly-language module.

No such file/directory

A file you specified in a command argument or in response to a prompt does not exist.

For instance, the message appears when you select Load from the File menu, and then enter the name of a nonexistent file.

No symbolic information

The program file you specified is not in the CodeView format.

You cannot debug in source mode, but you can use assembly mode.

Not a text file

You attempted to load a file using the Load selection from the File menu or using the View command, but the file is not a text file.

The CodeView debugger determines if a file is a text file by checking the first 128 bytes for characters that are not in the ASCII ranges 9–13 and 20–126.

Not an executable file

The file you specified to be debugged when you started the CodeView debugger is not an executable file having the extension **.EXE** or **.COM**.

Not enough space

You typed the Shell Escape command (!) or selected Shell from the File menu, but there is not enough free memory to execute **COMMAND.COM**.

Since memory is released by code in the FORTRAN start-up routines, this error always occurs if you try to use the Shell Escape command before you have executed any code. Use any of the code-execution commands (Trace, Program Step, or Go) to execute the FORTRAN start-up code, then try the Shell Escape command again. The message also occurs with assembly-language programs that do not specifically release memory.

Object too big

You entered a Tracepoint command with a data object (such as an array) that is larger than 128 bytes.

Operand types incorrect for this operation

An operand in a FORTRAN expression had a type incompatible with the operation applied to it.

For example, if P is declared as `CHARACTER P(10)`, then `? P+5` would produce this error, since a character array cannot be an operand of an arithmetic operator.

Operator must have a struct/union type

You used one of the C member-selection operators (`->` or `.`) in an expression that does not reference an element of a structure or union.

Operator needs lvalue

You specified an expression that does not evaluate to a memory location in an operation that requires one. (An lvalue is an expression that refers to a memory location.)

For example, `buffer(count)` is correct because it represents a symbol in memory. However, `I .EQV. 10` is invalid because it evaluates to TRUE or FALSE instead of a single memory location.

`Program terminated normally (`*number*`)`

You executed your program to the end. The number displayed in parentheses is the exit code returned to DOS by your program.

You must use the Restart command (or the Start menu selection) to start the program before executing more code.

`Radix must be between 2 and 36 inclusive`

You specified a radix outside the allowable range.

`Register variable out of scope`

You tried to specify a register variable using the period (.) operator and a routine name.

For example, if you are in a third-level routine, you can display the value of a local variable called `local` in a second-level routine called `parent` with the following command:

```
? parent.local
```

However, this command will not work if `local` is declared as a register variable.

`Regular expression too complex`

The regular expression specified is too complex for the CodeView debugger to evaluate.

`Regular expression too long`

The regular expression specified is too long for the CodeView debugger to evaluate.

`Simple variable cannot have argument`

In an expression, you specified an argument to a simple variable.

For example, given the declaration `INTEGER NUM`, the expression `NUM(I)` is not allowed.

Substring range out of bound

A character expression exceeds the length that is specified in the **CHARACTER** statement.

Syntax error

You specified an invalid command line for a dialog command.

Check for an invalid command letter. This message also appears if you enter an invalid assembly-language instruction using the Assemble command. The error will be preceded by a caret that points to the first character the CodeView debugger could not interpret.

Too few array bounds given

The bounds you specified in an array subscript do not match the array declaration.

For example, given the array declaration `INTEGER IARRAY(3,4)`, the expression `IARRAY(I)` would produce this message.

Too many array bounds given

The bounds you specified in an array subscript do not match the array declaration.

For example, given the array declaration `INTEGER IARRAY(3,4)`, the expression `IARRAY(I,3,J)` would produce this message.

Too many breakpoints

You tried to specify a 21st breakpoint; the CodeView debugger only permits 20.

Too many open files

You do not have enough file handles for the CodeView debugger to operate correctly.

You must specify more files in your **CONFIG.SYS** file. See your DOS user's guide for information on using the **CONFIG.SYS** file.

Type clash in function argument

The type of an actual parameter does not match the corresponding formal parameter.

This message also appears when a subroutine that uses alternate returns is called and the values of the return labels in the actual parameter list are not 0.

`Type conversion too complex`

You tried to type cast an element of an expression in a type other than the simple types or with more than one level of indirection.

An example of a complex type would be type casting to a struct or union type. An example of two levels of indirection would be `char **`.

`Unable to open file`

A file you specified in a command argument or in response to a prompt cannot be opened.

For instance, the message appears when you select Load from the File menu, and then enter the name of a file that is corrupted or has its file attributes set so that it cannot be opened.

`Unknown symbol`

You specified an identifier that is not in the CodeView debugger's symbol table.

Check for a misspelling. This message may also occur if you try to use a local variable in an argument when you are not in the routine where the variable is defined. The message also occurs when a subroutine that uses alternate returns is called and the values of the return labels in the actual parameter list are not 0.

`Unrecognized option` *option*
`Valid options: /B /C<command> /D /F /I /M /S /T /W /43 /2`

You entered an invalid option when starting the CodeView debugger.

Try retyping the command line.

`Usage: cv [options] file [arguments]`

You failed to specify an executable file when you started the CodeView debugger.

Try again with the syntax shown in the message.

`Video mode changed without /S option`

The program changed video modes (from or to one of the graphics modes) when screen swapping was not specified.

You must use the **/S** option to specify screen swapping when debugging graphics programs. You can continue debugging when you get this message, but the output screen of the debugged program may be damaged.

`Warning: packed file`

You started the CodeView debugger with a packed file as the executable file.

You can attempt to debug the program in assembly mode, but the packing routines at the start of the program may make this difficult. You cannot debug in source mode because all symbolic information is stripped from a file when it is packed with the **/EXEPACK** linker option or the **EXEPACK** utility.

`Wrong number of function arguments`

You specified an incorrect number of arguments for a function.

Glossary

Glossary

Address

A FORTRAN expression that evaluates to an address in memory. Addresses can be given in the *segment*:*offset* format. If the *segment* is not given, the default segment is assumed. The default segment is **CS** for commands related to code and **DS** for commands related to data.

Address range

A range of memory bounded by two addresses. The range can be specified in the normal format by giving the starting and ending addresses (inclusive), or it can be specified in the object-range format by specifying the starting address followed first by the letter "L" (uppercase or lowercase) and then by the number of objects in the range (#100 L 10, for example, specifies the range from #100 to #109, inclusive).

Assembly mode

The mode in which the CodeView debugger displays assembly-language-instruction mnemonics to represent the code being executed.

Basic input/output system (BIOS)

The code built into system memory that provides the lowest level of functionality in a computer system. You can trace into the BIOS with the CodeView debugger, using assembly mode.

Breakpoint

A specified address where program execution will be halted. The CodeView debugger stops whenever it reaches an address where a breakpoint has been set. *See* "Watchpoint" and "Tracepoint" for a description of conditional breakpoints.

Call to a routine

A call to a subroutine that performs a specific action. In FORTRAN (source mode), subroutines are called routines. In assembly language (assembly mode), subroutines are called procedures.

Click

To press and quickly release one of the buttons on the mouse while pointing the mouse at an object on the screen.

CodeView format specifier

A one-letter code that may be appended to an expression to control how the value of that expression will be formatted for display. These specifiers come from **printf**, a function in the standard library of the C language.

Color graphics adapter (CGA)

A video adapter capable of displaying text characters or graphics pixels. Color can also be displayed with the appropriate display monitor.

Cursor

The thin blinking line that represents the current location where the CodeView debugger is ready to accept commands. The cursor need not be in the dialog window to enter dialog commands.

Dialog box

A box that appears when you select a menu item that requires a response. The box asks you to enter some text. After you type your response and press the ENTER key (or a mouse button), the box disappears.

Dialog commands

Commands entered in the dialog window in window mode, or any command in sequential mode. Dialog commands consist of one- or two-character commands that can usually be followed by arguments.

Dialog window

The window at the bottom of the CodeView screen where dialog commands can be entered and previously entered dialog commands can be reviewed.

Display window

The window above the dialog window where source code is displayed in source mode or where assembly-language instructions are displayed in assembly mode.

Drag

To point the mouse at an object on the screen and press a mouse button, then while holding the button down, move the mouse. The object will move in the direction of the mouse movement. When the item is where you want it, release the button. The object will stay at that point.

Dump

To display the contents of memory at a specified memory location. In the CodeView debugger, the size of the object to be displayed is specified with a type character from the following list: **B** (Byte), **I** (Integer), **U** (Unsigned Integer), **W** (Word), **D** (Double Word), **S** (Short Real), **L** (Long Real), or **T** (10-Byte Real).

Enhanced graphics adapter (EGA)

A video adapter capable of displaying in all the modes of the color graphics adapter (CGA) plus many additional modes. The CodeView **/43** option displays in the EGA's 43-line text mode.

Flags

A register that controls the operation of many machine-level instructions. In other registers, the contents of the register are considered as a whole, while in the flags register only the individual bits have meaning. In the CodeView debugger, the current values of the most commonly used bits of the flags register are shown at the bottom of the register window.

Flipping

A screen-exchange method that uses the video pages of the CGA or EGA to store both the debugging and output screens. Video pages are areas of memory reserved for screen storage. When you request the other screen, the two video pages are exchanged. This method is faster than swapping, the other screen-exchange method, but it does not work with the monochrome adapter (MA) or with programs that do graphics or use the video pages.

Global symbol

A symbol that is available throughout the entire program. In the CodeView debugger, routine names are always global symbols. *See* "Local Symbol."

Highlight

The reverse-video line that marks the current selection on a menu. The highlight moves as you move the mouse or press the UP ARROW or DOWN ARROW key.

Identifier

A name that identifies a register or location in memory. The terms "identifier" and "symbol" are used synonymously in CodeView documentation.

Interrupt call

A machine-level procedure that can be called to execute a BIOS, DOS, or other function. You can trace into BIOS interrupts with the CodeView debugger, but not into the DOS interrupt.

Label

A symbol (identifier) representing an address in the code segment (**CS** register). In FORTRAN, a label is a number that identifies a line in the source code (marking, for example, the end of a **DO** loop or the target of a **GOTO** statement). The debugger does not recognize FORTRAN line labels. All references to labels in this manual refer to assembly-language labels.

Local symbol

A symbol that has value only within a particular function.

Menu bar

The bar at the top of the CodeView display containing menu titles and the titles Trace! and Go!.

Message box

A box that pops up and displays a message. In window mode, message boxes are used to display error messages. You can press any key or a mouse button to make the box disappear.

Monochrome adapter (MA)

A video adapter capable of displaying only in black and white. Most monochrome adapters display text only; individual graphics pixels cannot be displayed. The CodeView debugger recognizes monochrome adapters and automatically selects swapping as the screen-exchange mode.

Mouse

A small pointing device designed to fit comfortably under your hand. By moving it about on a flat surface, you can move the mouse pointer in the corresponding direction on the screen.

Object range

See "Address range."

Output screen

The screen where program output is shown. The Screen Exchange command (\), the Output selection from the View menu, and the F4 key can be used to switch to this screen. The output screen is the same as it would be if you ran the debugged program outside of the CodeView debugger.

Pointer

The reverse-video square that moves to indicate the current position of the mouse. The mouse pointer appears only if a mouse is installed. To select an item with the mouse, move the mouse until the pointer rests on the item.

Pop-up menu

A menu that pops up when you point the mouse cursor to the menu title and press a mouse button. In the CodeView debugger, pop-up menus also pop up when you press the ALT key and the first letter of the menu title at the same time. You can make a selection from the menu by dragging the highlight up or down with the mouse, by pressing the UP ARROW or DOWN ARROW key to move the highlight, or by pressing the ALT key and the first letter of the selection title at the same time.

printf

A function in the C standard library that prints formatted output according to instructions supplied with a type-specifier argument. The CodeView debugger uses a subset of the **printf** type specifiers to format expression values. *See also* "CodeView format specifier."

Procedure call

A call to a subroutine that performs a specific action. In assembly language (assembly mode), subroutines are called procedures.

Program Step

To trace the next source line in source mode, or the next instruction in assembly mode. If the source line or instruction contains a routine, procedure, or interrupt call, the call is executed to the end and the CodeView debugger is ready to execute the instruction after the call. *See also* "Trace."

Radix

The number system in which numbers are specified. In the CodeView debugger, numbers can be entered in three radixes: 8 (octal), 10 (decimal), or 16 (hexadecimal). The default radix is 10.

Redirection

To specify the device from which a program will receive input or to which it will send output. Normally program input comes from the keyboard, and program output goes to the screen. Redirection involves specifying a device (or file) other than the default device. In the DOS operating system, input is redirected with the less-than symbol (<) and output is redirected with the greater-than symbol (>). The same symbols are used in the CodeView debugger to redirect input or output of the debugging session. In addition, the equal sign (=) can be used to redirect both input and output.

Register window

The optional window in which the central processing unit (CPU) registers and the bits of the flag register are displayed.

Registers

The places in memory where byte- or word-sized data can be stored during machine-level processing. The registers used with the 8086 family of processors are **AX**, **BX**, **CX**, **DX**, **SP**, **BP**, **SI**, **DI**, **DS**, **ES**, **SS**, **CS**, **IP**, and the flags register. *See also* "Flags."

Regular expressions

A system of specifying text patterns that match variable text strings. Regular expressions can be used to find strings in source files.

Routine

Any block of code that is called from somewhere else in a program. After the routine is executed, program control returns to the statement following the calling statement. In FORTRAN, subprograms (subroutines or functions) are routines. Intrinsic functions are also routines

(the call takes place on the assembly-language level to a routine in the FORTRAN libraries). Assembly-language routines are also called procedures.

Screen Exchange

The method by which both the output screen and the debugging screen are kept in memory so that both can be updated simultaneously and either viewed at the user's convenience. The two screen-exchange modes are flipping and swapping. *See* "Flipping" and "Swapping."

Sequential mode

The mode in which all CodeView output is sequential and no windows are available. Input and output scroll down the screen and the old output scrolls off the top of the screen when the screen is full. You cannot examine previous commands after they scroll off the top. This mode is required with computers that are not IBM compatible. The mouse and most window commands are not supported in sequential mode. Any debugging operation that can be done in window mode can also be done in sequential mode.

Shell Escape

A method of leaving the CodeView debugger without losing the current debugging context. You can "escape to a shell," do various DOS tasks, and then return to the debugger. The debugging screen will be the same as when you left it. The CodeView debugger creates the shell by saving all current operations to memory and invoking a second copy of **COMMAND.COM**.

Source mode

The mode in which the CodeView debugger displays FORTRAN source code to represent the code being executed.

Stack Trace

A symbolic representation of the functions that have been executed to reach the current instruction address. As a function is executed, the function address and any function arguments are pushed on the stack (the area of memory starting at the address of the **SS** register). Therefore, a trace of the stack always shows the currently active functions and the values of their arguments.

Start-up code

The code that the compiler places at the beginning of every program to control execution of the program code. When the CodeView debugger is loaded, the first source line executed runs the entire start-up code. If you switch to assembly mode before executing any code, you can trace through the start-up code.

Swapping

A screen-exchange method that uses buffers to store the debugging and output screens. When you request the other screen, the two buffers are exchanged. This method is slower than flipping, the other screen-exchange method, but it works with any adapter and any type of program.

Symbol

A name that identifies a location in memory. In FORTRAN, the names of variables and subprograms are symbols. The terms "symbol" and "identifier" are used synonymously in CodeView documentation.

Toggle

A function key or menu selection that turns a feature off if it is on, or on if it is off. When used as a verb, toggle means to reverse the status of a feature. For example, the F3 key is a toggle that switches between source and assembly modes. Press the F3 key to toggle either mode.

Trace

To trace the next source line in source mode, or the next instruction in assembly mode. If the source line or instruction contains a routine, procedure, or interrupt call, the first source line or instruction of the call is executed. The CodeView debugger is ready to execute the next instruction inside the call. *See also* "Program Step."

Tracepoint

A variable breakpoint that is taken when a value at a specified memory location changes. The value to be tested can be either the value of a CodeView expression, or any of the values in a range of memory. Tracepoints can slow program execution significantly, since the CodeView debugger has to check after executing each source line in source mode or after each instruction in assembly mode to see if the value has changed.

Watch window

The window where watch statements and their values are displayed. The three kinds of watch statements are watch expressions, watchpoints, and tracepoints.

Watchpoint

A variable breakpoint that is taken when a specified expression becomes nonzero (true). Watchpoints can slow program execution significantly, since the CodeView debugger has to check after executing each source line in source mode or after each instruction in assembly mode to see if the value is true.

Window commands

Commands that work only in the CodeView debugger's window mode. Window commands consist of function keys, mouse selections, CONTROL- and ALT-key combinations, and selections from pop-up menus.

Window mode

The mode in which the CodeView debugger displays separate windows that can change independently. The debugger has mouse support and a wide variety of window commands in window mode.

Index

Index

* (asterisk)
  Comment command, 211, 225
  multiplication operator, 75
  regular expressions, used in, 234
** (asterisks), exponentiation operator, 75
@ (at sign)
  Redraw command, 199, 226
  register prefix, 80
\ (backslash), Screen Exchange command
  described, 220
  dialog command, 226
  output screen, switch to, 253
  using, 200
{ } (braces), 8
[ ] (brackets)
  notational conventions, 8
  regular expressions, used in, 233
^ (caret), regular expressions, used in, 234, 235
: (colon)
  Delay command, 212, 225
  operator, 75, 81
– (dash)
  option designator, 16, 23
  regular expressions, used in, 233
$ (dollar sign), regular expressions, used in, 235
= (equal sign)
  assignment operator, 75
  Redirection command, 210, 226
! (exclamation point)
  command indicator, 39
  Shell Escape command, 205, 226
> (greater-than sign)
  CodeView prompt, 39, 40, 71
  Redirection command, 209, 226
< (less-than sign), redirected input command, 208, 226
– (minus sign), 75
(number sign), Tab Set command, 207, 226
( ) (parentheses), 75
. (period)
  Current Location command, 169, 225
  operator, 75, 244
  regular expressions, used in, 232
+ (plus sign), 75
? (question mark), Display Expression command, 225
" (quotation marks)
  notational conventions, 9
  Pause command, 213, 226
/ (slash)
  division operator, 75
  option designator, 16, 23
  Search command, 203, 226
_ (underscore), symbol names, used in, 78
| (vertical bar), notational conventions, 8

10-byte reals
  dumping, 121
  entering, 189
/2 CodeView option, 32
/43 CodeView option, 32, 221, 251
7 (8087 command), 124, 227
8087
  command, 123, 227
  coprocessor, 124, 178
8259 masking, 31

A (Assemble command), 176, 225, 245
Absolute addresses, 81
Adapters, using two, 32
Address ranges, arguments, used as, 82, 249
Addresses
  absolute, 143
  arguments, used in, 81, 237, 242, 249
  full, 81, 143
.AND. operator, 75

Arguments
  dialog commands
    entering, 71
    errors, 241, 245
    format, 73
  function, 255
  program, 20, 100
  routine, 64, 170
Arithmetic operator
  addition (+), 75
  division (/), 75
  exponentiation (**), 75
  multiplication (*), 75
  subtraction (-), 75
ASCII characters, 114, 115
Assemble command, 175, 225, 245
Assembly address, 176
Assembly language, 4
Assembly mode
  defined, 249
  described, 220
  display options, 60
  example, 165
  setting, 161
  using, 33, 242
Assembly rules, 176
Assignment operator, 75
Asterisk (*)
  Comment command, 211, 225
  multiplication operator, 75
  regular expressions, used in, 234
Asterisks (**), exponentiation operator, 75
At sign (@)
  Redraw command, 199, 226
  register prefix, 80

/B CodeView option, 9, 25, 221
Backslash (\), Screen Exchange command
  described, 220
  dialog command, 226
  output screen, switch to, 253
  using, 200
BACKSPACE key, 72
Basic input/output system (BIOS), 249
BC (Breakpoint Clear command), 133, 225, 237, 239
BD (Breakpoint Disable command), 134, 225, 237, 239
BE (Breakpoint Enable command), 136, 225, 237, 239
BIOS (basic input/output system), 249
BL (Breakpoint List command), 137, 225
Black-and-white display, 9, 25, 221
Bold type, notational conventions, 6
BP (Breakpoint Set command), 130, 225, 242, 245
Brackets ([ ])
  notational conventions, 8
  regular expressions, 233
"Break when" point, 227, 237
Breakpoint Disable command, 134, 225, 237, 239
Breakpoint Enable command
  argument requirements, 237, 239
  listing, 225
  using, 135
Breakpoint List command, 136, 225
Breakpoint Set command
  dialog command, 225
  F9 function key, 43, 67
  mouse, executing with, 47
  using, 129
  window command, 222
Breakpoints
  address, 96
  conditional, 59, 129, 141
  defined, 129, 249
  deleting, 132, 225
  displaying, 39, 130
  Go command, used with, 95
  listing, 136, 225
Buffer, command, 41, 72

/C CodeView option, 26, 221
C, Options menu selection, 63
Call, defined, 11
Calling conventions, 170
Calls, 90, 93
Calls menu, 63, 171, 223, 255
Capital letters, notational conventions, 6, 9
Caret (^), regular expressions, used in, 234, 235
Case sensitivity, 62, 73, 78, 223

CGA (color graphics adapter), 250
Clear Breakpoints command
  argument requirements, 237, 239
  dialog command, 225
  Run menu selection, 57, 137, 223
  using, 132
Click, defined, 46, 250
CodeView format prefixes, 106, 228, 238
CodeView format specifiers
  defined, 11, 250
  Evaluate selection, used with, 56
  summary, 105
  table, 228
  tracepoint command, used with, 151
  watchpoint commands, used with, 144, 147
  watch expression statement, used with, 144
/CODEVIEW linker option, 18, 19, 33
Colon (:)
  Delay command, 212, 225
  operator, 75, 81
Color graphics adapter (CGA), 28, 32, 250
.COM extension, debugged files, used for, 20, 33, 243
Command buffer, 41, 72
COMMAND.COM, Shell command, used with, 52, 204
Commands
  8087, 123, 227
  Assemble, 175, 225, 245
  Breakpoint Disable, 134, 225, 237, 239
  Breakpoint Enable, 135, 225, 237, 239
  Breakpoint List, 136, 225
  Breakpoint Set
    dialog command, 225
    F9 function key, 43, 67
    mouse, executing with, 47
    using, 129
    window command, 222
  Clear Breakpoints
    argument requirements, 237, 239
    dialog command, 225
    Run menu selection, 57, 137, 223
    using, 132
  Comment, 211, 225
Commands *(continued)*
  Current Location, 169, 225
  Delete Watch, 59, 155, 223, 227
  Delay, 212, 225
  dialog, 39, 40, 71, 250
  Display Expression, 56, 103, 225
  dump
    default size, 112, 114
    summary, 225
  Dump
    10-Byte Reals, 121
    ASCII, 115
    Bytes, 114
    Double Words, 118
    Integers, 116
    Long Reals, 120
    Short Reals, 119
    Unsigned Integers, 117
    Words, 117
  enter, summary, 225
  Enter
    10-Byte Reals, 189
    ASCII, 183
    Bytes, 183
    default size, 182
    Double Words, 186
    Integers, 184
    Long Reals, 188
    Short Reals, 187
    Unsigned Integers, 185
    Words, 186
  Examine Symbols, 109, 225
  Execute, 57, 98, 223, 225
  expression, 56, 103, 223, 225
  Go
    dialog command, 226
    F5 function key, 43, 67
    mouse, executing with, 49
    using, 95
    window command, 222
  Goto
    dialog command, 226
    F5 function key, 43
    mouse, executing with, 48
    using, 95
    window command, 222
  Help
    dialog command, 226
    F1 function key, 42, 223
    screen mode, 220

Commands *(continued)*
Help *(continued)*
sequential mode, 66
using, 65, 195
View menu selection, 55
window mode, 65, 222
Load, 226
move cursor down, 41, 222
move cursor up, 41, 222
move separator line down, 41, 46, 222
move separator line up, 41, 46, 222
Pause, 213, 226
Program Step
defined, 254
dialog command, 226
F10 function key, 44, 67
mouse, executing with, 48
using, 93
window command, 222
Quit, 53, 196, 223, 226
Radix
defined, 254
limits, 238
setting, 79, 226
using, 197
redirected input, 208, 226
redirected input and output, 210, 226
redirected output, 209, 226
redirection, 207, 226, 254
Redraw, 199, 226
Registers
changing register values, 190
dialog command, 226
displaying registers, 122
F2 function key, 42, 66
mouse, executing with, 49
Options menu selection, 62
window command, 222
Restart
dialog command, 226
Run menu selection, 57, 223
using, 99
Screen exchange
defined, 253
described, 220
dialog command, 200, 226
F4 function key, 42, 67
output screen, switching to, 255
View menu selection, 56, 222
window command, 226
Commands *(continued)*
scroll line down, 47
scroll line up, 47
scroll page down, 41, 47, 222
scroll page up, 41, 47, 222
scroll to bottom, 42, 47, 222
scroll to top, 41, 47, 222
Search
dialog command, 226
menu selections, 53, 223
regular expressions, used with, 231
using, 201
Set mode
described, 220
dialog command, 161, 226
F3 function key, 42, 66
View menu selection, 55, 223
window command, 222
Shell escape
defined, 255
dialog command, 203, 226
File menu selection, 52, 223
space problem solutions, 243
Stack Trace
Calls menu selection, 223
defined, 255
dialog command, 226
display contents, 64
using, 170
Tab Set, 206, 226
Trace
defined, 256
dialog command, 227
F8 function key, 43, 67
mouse, executing with, 48
using, 90
window command, 222
Tracepoint
data object size limit, 243
defined, 256
dialog command, 227
format specifier, used with, 227
sequential mode, 68
using, 150
Watch menu selection, 59, 223
Unassemble, 163, 227
Use, 227
View, 166, 227, 242
Watch
deleting watch statements, 155

Commands *(continued)*
Watch *(continued)*
dialog command, 227
menu selections, 58, 223
sequential mode, 68
using, 142, 227, 238
Watch List, 68, 156, 227
Watchpoint
defined, 257
deleting, 155
dialog command, 227
format specifier, 238
sequential mode, 68
setting, 146
Watch menu selection, 59, 223
Comment
command, 211, 225
line, 95, 96, 130, 131
Compiler errors, 17
COMSPEC environment variable, 204
Conditional breakpoints, 59, 129, 141
CONFIG.SYS file, 245
Conjunction operator, 75
Constant numbers, arguments, used as, 78, 240
CONTROL-BREAK, 31, 44, 90, 150
CONTROL-C, 31, 44, 71, 90, 150
CONTROL-D (separator line down), 41, 222
CONTROL-S, 72
CONTROL-U (separator line up), 41, 222
/CPARMAXALLOC link option, 204
Current Location command, 169, 225
Current location line, 39
Cursor, 39, 71, 250
CV.EXE, location of, 19
CV.HLP, location of, 19, 65

/D CodeView option, 31
D (Dump command), 114
DA (Dump ASCII command), 115
Dash (–)
option designator, 16, 23
regular expressions, 233
DB (Dump Bytes command), 114
DD (Dump Double Words command), 118
DEBUG, 4, 37, 66
Debugging modes, 220
Default
address-range size, 113
assembly-mode format, 60
expression format, 144
IBM Personal Computer, used with, 23
radix, 170, 197, 198, 254
segment, 81
start-up behavior, 21
type
dialog commands, 224
dump commands, 114
enter commands, 182
trace commands, 153
watch commands, 145
Delay command, 212, 225
Delete Watch command, 59, 155, 223, 227
Destination address, Go command, used with, 95
DI (Dump Integers command), 116
Dialog
box, 40, 45, 50, 250
window, 39, 250
Disjunction, inclusive, 75
Display
mode, 89, 165, 167, 220
window, 39, 250
Display Expression command, 56, 103, 225
Dividing by zero, 240
DL (Dump Long Reals command), 120
Dollar sign ($), regular expressions, used in, 235
Down arrow key (cursor down), 41, 222
Drag, defined, 46, 251
DS (Dump Short Reals command), 119
DT (Dump 10-Byte Reals command), 121
DU (Dump Unsigned Integers command), 117
Dump address, 113
Dump commands
10-Byte Reals, 121
ASCII, 115
Bytes, 114
default size, 114
Double Words, 118
Integers, 116
Long Reals, 120

Dump commands *(continued)*
Short Reals, 119
summary, 225
Unsigned Integers, 117
using, 112
Words, 117
Dump, defined, 251
DW (Dump Words command), 117

E
Enter command, 182
Execute command, 98, 225
EA (Enter ASCII command), 183
EB (Enter Bytes command), 183
Echo, redirection, used with, 209
ED (Enter Double Words command), 186
EGA (enhanced graphics adapter), 251
EI (Enter Integers command), 184
EL (Enter Long Reals command), 188
Ellipses, notational conventions, 8
END key
exit help, 66
scroll to bottom, 42, 222
Enhanced graphics adapter (EGA), 28, 32, 251
Enter commands
ASCII, 183
Bytes, 183
default size, 182
Double Words, 186
Integers, 184
Long Reals, 188
Short Reals, 187
summary, 225
10-Byte Reals, 189
Unsigned Integers, 185
using, 178
Words, 186
.EQ. operator, 75
Equal sign (=)
assignment operator, 75
Redirection command, 210, 226
Equal-to operator, 75
Equivalence operator, 75
.EQV. operator, 75
Error
internal debugger, 238, 240
logical, 17
Error *(continued)*
messages, 237
syntax, 17
Errorlevel code, 95
ES (Enter Short Reals command), 187
ESCAPE key, 45, 66
ET (Enter 10-Byte Reals command), 189
EU (Enter Unsigned Integers command), 185
Evaluate, menu selection, 106
EW (Enter Words command), 186
Examine Symbols command, 109, 225
Exchange modes, 220
Exclamation point (!)
command indicator, 39
Shell Escape command, 205, 226
.EXE extension, debugged files, used for, 20, 21, 33, 243
Executable file
CodeView format, 15, 18
command line, used in, 20, 239, 243
location of, 19
start-up, required for, 21
Execute command, 57, 98, 223, 225
/EXEPACK link option, 247
Exit code, 95, 97
Exit, DOS command, 52, 204
Expression evaluation
dialog command, 225
Display Expression command, 103
expression requirements, 74
syntax requirements, 241
View menu selection, 56, 223
Expression evaluator, selecting, 63
Expressions
arguments
error in, 240, 241
used as, 74
regular
defined, 254
limits, 244
searches, used in, 54, 201, 242
special characters, 231
specifying, 232, 233, 234, 235

/F CodeView option, 27, 220, 221
F1 key (Help), 42, 65, 66, 196, 220, 222
F2 key (Register), 42, 66, 122, 190, 222

F3 key (Set source/assembly), 42, 66, 162, 220, 222
F4 key (Screen Exchange), 42, 67, 200, 220, 222, 253
F5 key (Go), 43, 51, 67, 96, 222
F6 key (switch cursor), 40, 96, 222
F7 key (Goto), 43, 96, 222
F8 key (Trace), 43, 51, 67, 91, 222
F9 key
  Breakpoint Clear, 133
  Breakpoint Set, 43, 67, 135, 222
F10 key (Program Step), 44, 51, 67, 93, 222
Far-return mnemonic (RETF), 176
File handles, 245
File menu
  Load
    described, 51
    error, 242, 246
    table, 223
    View command, 166
  Quit, 53, 196, 223
  Shell, 52, 204, 223, 243
FL compiler control program, 16, 17, 19
Flag bits
  changing with the mouse, 49
  defined, 251
  values
    changing, 190
    displaying, 122
Flag mnemonics, 191, 237
Flipping, screen, 27, 220, 221, 251
Format specifiers
  Evaluate selection, used with, 56
  summary, 105
  table, 228
  tracepoint commands, used with, 151
  watchpoint commands, used with, 144, 147
FORTRAN operators, 74
FORTRAN, Options menu selection, 63
Function
  keys
    CONTROL-D (separator line down), 41, 222
    CONTROL-U (separator line up), 41, 222
    down arrow (cursor down), 41, 222
    END (exit help), 66
  keys *(continued)*
    END (scroll to bottom), 42, 222
    ESC (exit help), 66
    F1 (Help), 42, 65, 66, 196, 220, 222
    F2 (Register), 42, 66, 122, 190, 222
    F3 (Set source/assembly), 42, 66, 220, 222
    F4 (Screen Exchange), 42, 67, 200, 220, 222, 253
    F5 (Go), 43, 51, 67, 96, 222
    F6 (switch cursor), 40, 96, 222
    F7 (Goto), 43, 96, 222
    F8 (Trace), 43, 51, 67, 91, 222
    F9
      Breakpoint Clear, 133
      Breakpoint Set, 43, 67, 135, 222
    F10 (Program Step), 44, 51, 67, 93, 222
    HOME
      scroll
      top of help, 65
    PGDN
      next help, 65
      scroll page down, 41, 222
    PGUP
      previous help, 65
      scroll page up, 41, 222
    up arrow (cursor up), 41, 222
Functions
  calls, 64, 255
  examining, 109
  intrinsic, 77
  viewing, 64

G (Go command), 96, 226
.GE. operator, 75
Go command
  dialog command, 226
  F5 function key, 43, 67
  mouse, executing with, 49
  using, 95
  window command, 222
Goto command
  dialog command, 226
  F5 function key, 43
  mouse, executing with, 48
  using, 95
  window command, 222
Graphics programs, debugging, 32, 209

Greater-than operator, 75
Greater-than sign (>)
  CodeView prompt, 39, 40, 71
  Redirection command, 209, 226
Greater-than-or-equal-to operator, 75
.GT. operator, 75

H (Help command), 196, 220, 226
Help command
  dialog command, 226
  F1 function key, 42, 223
  screen mode, 220
  sequential mode, 66
  using, 195
  view menu selection, 55
  window mode, 65, 222
Highlight, 39, 252
HOME key
  scroll to top, 41, 222
  top of help, 65
Hyphen (-), 75

/I CodeView option, 31
IBM PC
  CodeView compatibility with, 3, 28, 29
  recognizing, 23
Identifiers, arguments, used as, 246, 252, 256
Immediate operand, 177
Include files, 16
Inclusive disjunction operator, 75
#IND (indefinite), 113
IND (indefinite), 113
Indentation, 206
Indirect register instructions, 177
#INF (infinity), 113
INF (infinity), 113
Infinity, 113
Instruction, current, 90, 93
Instruction-name synonyms, 178
Integers, dumping, 116
Interrupt
  calls, 252
  DOS functions, 90
Intrinsic functions, 77
Italics, notational conventions, 7

K (Stack Trace command), 171, 226
Key names, notational conventions, 9

L (Load command), 226
L (Restart command), 100, 226, 242, 244
Labels
  defined, 10
  finding, 55, 202, 223
.LE. operator, 75
Less-than operator, 75
Less-than sign (<), redirected input command, 208, 226
Less-than-or-equal-to operator, 75
LIB, CodeView, used with, 17
Line numbers, arguments, used as, 83
LINK (object linker), 16, 18
Load command, 226
Load, menu selection, 100
Logical
  error, 17
  operator
    .AND., 75
    .EQV., 75
    .NEQV., 75
    .NOT., 75
    .OR., 75
Long reals
  dumping, 120
  entering, 188
Loops
  tracepoints, used with, 154
  watchpoints, used with, 150
.LT. operator, 75

/M CodeView option, 30, 221
MA (monochrome adapter). *See* Monochrome adapter
Macro Assembler, 5, 33
Memory release, 204, 243, 255
Menu bar, 39, 252
Menus
  Calls
    Stack Trace command, used with, 171
    using, 63, 223, 255
  defined, 39

Menus *(continued)*
File
Load, 51, 166, 223, 246
Quit, 53, 196, 223
Shell, 52, 204, 223
keyboard, selecting with, 44
mouse, selecting with, 49
Options
Bytes Coded, 62, 162, 223
C, 223
Case Sense, 62, 223
Flip/Swap, 61, 223
FORTRAN, 223
Mix Source, 61, 162, 223
Registers, 62, 122, 190, 223
Symbols, 62, 162, 223
Run
Clear Breakpoints, 57, 133, 223
Execute, 57, 98, 223
Restart, 57, 99, 223, 244
Start, 57, 99, 223, 244
Search
Find, 53, 202, 223
Label, 55, 202, 223
Next, 54, 202, 223
Previous, 54, 202, 223
View
Assembly, 56, 162, 220, 223
Evaluate, 56, 106, 223
Help, 55, 65, 195, 220, 223
Output, 56, 200, 220, 223
Source, 55, 162, 220, 223
Watch
Add Watch, 58, 144, 223
Delete Watch, 59, 155, 223
Tracepoint, 59, 151, 223, 256
Watchpoint, 59, 147, 223, 257
Message box, 40, 45, 50, 252
Minus sign (−), 75
Mixed mode, 161, 220
Modules, examination, 109
Monochrome adapter (MA), 28, 32, 252
Mouse
compatibility, 4
defined, 253
driver, 31
ignore option, 30, 221
pointer, 40, 46
selecting with, 46

N (Radix command), 198, 226, 238
NAN (not a number), 113
.NE. operator, 75
Negation operator, 75
.NEQV. operator, 75
NMI trapping, 31
Nonequivalence operator, 75
Nonproportional typeface, notational conventions, 7
.NOT. operator, 75
Notational conventions
bold type, 6
brackets, 8
capital letter, 6
ellipses, 8
italics, 7
listed, 7
nonproportional typeface, 7
quotation marks, 9
sample screens, 9
vertical bar, 8
Not-equal-to operator, 75
Number sign (#), Tab Set command, 207, 226
Numbers
arguments, used as, 78, 240
floating point, 119, 120, 121

Object ranges, arguments, used as, 82, 253
/Od compiler option, 17
Operands, 123
Operators
addition (+), 75
CodeView subset, 74
division (/), 75
exponentiation (**), 75
logical
.AND., 75
.EQV., 75
.NEQV., 75
.NOT., 75
.OR., 75
multiplication (*), 75
relational
.EQ., 75
.GE., 75
.GT., 75
.LE., 75

relational *(continued)*
.LT., 75
.NE., 75
subtraction (−), 75
Optimization, 17
Optional fields, conventions for, 8
Options
CodeView
/2, 32
/43, 32, 221, 251
/B, 25, 221
/C, 26, 221
command line, used in, 20
/D, 31
/F, 27, 220, 221
/I, 31
/M, 30, 221
/S, 27, 220, 221, 246
summary, 23
/T, 29, 220, 221
/W, 29, 220, 221
compiler
/Od, 17
/Zd, 17
/Zi, 17
linker
/CODEVIEW, 18, 19, 33
/CPARMAXALLOC, 204
/EXEPACK, 247
menu
Bytes Coded, 62, 162, 223
C, 63, 223
Case Sense, 62, 73, 223
Flip/Swap, 61, 223
FORTRAN, 63, 223
Mix Source, 61, 162, 223
Registers, 62, 122, 190, 223
Symbols, 62, 162, 223
.OR. operator, 75
Output screen, 27, 200, 253, 255
Overlays, 20

P (Program Step command), 93, 226
Parameters, program, 20
Parentheses ( ), 75
Pass count, 131, 137
PATH command, 19
Pause command, 213, 226
Period (.)
Current Location command, 169, 225
operator, 75, 244
regular expressions, used in, 232
PGDN (next help), 65
PGDN (scroll page down), 41, 168, 222
PGUP (previous help), 65
PGUP (scroll page up), 41, 222
Plus sign (+), 75
Point, defined, 46
Pointer, mouse, 40, 46, 253
Pop-up menu, defined, 253
Precedence of operators, 74
Prefixes, format specifiers, used with, 106, 228, 238
Procedure calls, 90, 93, 253
Procedures, 109, 170, 255
Program Step command
defined, 254
dialog command, 226
F10 function key, 44, 67
mouse, executing with, 48
using, 93
window command, 222
Prompt, CodeView, (>) greater-than sign, 39, 40, 71
Protected-mode (80286) mnemonics, 163, 165, 175
Public symbols, 33

Q (Quit command), 197, 226
Question mark (?), Display Expression command, 225
Quit command, 53, 196, 223, 226
Quotation marks (")
notational conventions, 9
Pause command, used as, 213, 226

R (Register command), 122, 190, 226, 237, 238
Radix
command
defined, 254
limits, 238
setting, 79, 226
using, 197
current, 64, 78, 170
Ranges, arguments, used as, 82

README.DOC file, 6
Redirection
  commands, 207, 226, 254
  redirected input and output command, 210, 226
  redirected input command, 208, 226
  redirected output command, 209, 226
  start-up commands, used in, 26
Redraw command, 199, 226
Registers
  argument, used as, 80
  command
    changing register values, 190
    dialog command, 226
    displaying registers, 122
    F2 function key, 42, 66
    mouse, executing with, 49
    Options menu selection, 62
    window command, 222
  variables, 244
  window, 39, 254
Regular expressions
  defined, 254
  limits, 244
  searches, used in, 54, 201, 242
  specifying, 231
Relational
  expressions, 147
  operators
    .EQ., 75
    .GE., 75
    .GT., 75
    .LE., 75
    .LT., 75
    .NE., 75
Restart command
  dialog command, 226
  Run menu selection, 57, 223
  using, 99, 242, 244
ROM (read-only memory), 91
Routines
  arguments, value of, 170
  calls to, 171, 249
  defined, 10, 254
  keys
    F3 (Set source/assembly), 162
    PGDN (scroll page down), 168
Run menu
  Clear Breakpoints, 57, 133, 223
  Execute, 57, 98, 223
Run menu *(continued)*
  Restart, 57, 99, 223, 244
  Start, 57, 99, 223, 244

/S CodeView option, 27, 220, 221, 246
S (Set Mode command), 162, 220, 226
Sample session, described, 15
Screen
  buffer, 143
  exchange method, 27
  modes, 220
  movement commands, 41, 222
  notational conventions, 9
  two, using, 32
Screen exchange
  defined, 253
  described, 220
  dialog command, 200, 226
  F4 function key, 42, 67
  output screen, switching to, 255
  View menu selection, 56, 222
  window command, 223
Scroll bar, defined, 39
Search command
  dialog command, 226
  menu selections, 53, 223
  regular expressions, used with, 231
  using, 201
Search menu
  Find, 53, 202, 223
  Label, 55, 202, 223
  Next, 54, 202, 223
  Previous, 54, 202, 223
Separator line, 39
Sequential mode
  defined, 255
  redirection, used with, 210
  required when, 37
  starting, 29, 220, 221
  using, 66
Set Block, DOS function call (#4A), 204
Set mode
  described, 220
  dialog command, 161, 226
  F3 function key, 42, 66
  View menu selection, 55, 223
  window command, 222

Shell escape command
  defined, 255
  dialog command, 203, 226
  File menu selection, 52, 223
  space problem solutions, 243
Short reals
  dumping, 119
  entering, 187
Slash (/)
  division operator, 75
  option designator, 16, 23
  Search command, 203, 226
Small capitals, notational conventions, 9
Source
  code, writing, 16
  file, line-number arguments, used with, 83
  mode, 161, 220, 242, 255
Source-module files, location of, 20, 52
Stack, 8087, 125
Stack Trace command
  Calls menu selection, 223
  defined, 255
  dialog command, 226
  display contents, 64
  using, 170
Start-up
  code, 22, 52, 204, 256
  command line, 20, 239, 243
  commands, 26, 221
  file configuration, 19
String mnemonics, 176
Strings, arguments, used as, 84, 241
Style, programming, 16
Swapping, screen, 27, 220, 221, 256
Symbols
  arguments, used in, 78, 246, 252
  defined, 10, 256
  examining, 109
  names, spelling, 78
SYMDEB, 4, 37, 66
Syntax
  error, 17
  summary, 195, 226
Syntax conventions. *See* Notational conventions
SYSTEM-REQUEST key, 44, 90

/T CodeView option, 29, 220, 221
T (Trace command), 91, 227
Tab Set command, 206, 226
Text files, identifying, 242
Text strings, finding, 53, 201, 223, 226, 231
Toggle, defined, 256
TP (Tracepoint command), 152, 227, 237, 243
Trace command
  defined, 256
  dialog command, 227
  F8 function key, 43, 67
  mouse, executing with, 48
  using, 90
  window command, 222
Tracepoint
  command
    data object size limit, 243
    defined, 256
    dialog command, 227
    format specifier, used with, 227, 237
    sequential mode, 68
    using, 150
    Watch menu selection, 59, 223
  defined, 150, 256
Two-color graphics display, 25, 221
Type casting, 238
Type conversion, 76

U (Unassemble command), 164, 227
Unassemble command, 163, 227
Underscore (_), symbol names, used in, 78
Unsigned integers, dumping, 117
Up arrow key (cursor up), 41, 222
Uppercase letters, notational conventions, 6
Use command, 227

V (View command), 167, 227, 242
Variables, local, 17, 76, 142, 252
Vertical bar (|), notational conventions, 8
Video modes, 246
Video-display pages, 27

View
  command, 166, 227, 242
  menu
    Assembly, 56, 162, 220, 223
    Evaluate, 56, 106, 223
    Help, 55, 65, 195, 220, 223
    Output, 56, 200, 220, 223
    Source, 55, 162, 220, 223

/W CodeView option, 29, 220, 221
W (Watch command), 144, 227, 238
W (Watch List command), 157, 227
WAIT instruction, 178
Watch
  expression statement, 143
  memory statement, 144
  menu
    Add Watch, 58, 144, 223
    Delete Watch, 59, 155, 223
    Tracepoint, 59, 151, 223, 256
    Watchpoint, 59, 147, 223, 257
  statement
    commands, 141
    defined, 39
    deleting, 155, 227
    described, 141
    listing, 156, 227
  window, 39, 141
Watch command
  deleting watch statements, 155
  dialog command, 227
  format specifier, used with, 238
  menu selections, 58, 223
  sequential mode, 68
  using, 142
Watch List command, 68, 156, 227
Watchpoint
  command
    defined, 257
    dialog command, 227
    format specifier, used with, 238
    menu selections, 223
    sequential mode, 68
    setting, 146
    Watch menu selection, 59, 227
  defined, 146, 227, 238, 257
Window
  commands, 40, 71, 257
Window *(continued)*
  mode
    defined, 257
    required when, 37
    starting, 29, 220, 221
WP (Watchpoint command), 147, 227, 238

X (Examine Symbols command), 109, 225

Y (Watch Delete command), 155, 227

/Zd compiler option, 17
Zero, dividing by, 240
/Zi compiler option, 17

MICROSOFT®
16011 NE 36th Way, Box 97017, Redmond, WA 98073-9717

Software Problem Report

Name _____

Street _____

City _____ State _____ Zip _____

Phone _____ Date _____

Instructions

Use this form to report software bugs, documentation errors, or suggested enhancements. Mail the form to Microsoft.

Category

_____ Software Problem

_____ Software Enhancement

_____ Documentation Problem
(Document #_____)

_____ Other

Software Description

Microsoft Product _____

Rev. _____ Registration # _____

Operating System _____

Rev. _____ Supplier _____

Other Software Used _____

Rev. _____ Supplier _____

Hardware Description

Manufacturer _____ CPU _____ Memory _____ KB

Disk Size _____" Density: Sides:

Single _____ Single _____

Double _____ Double _____

Peripherals _____

Problem Description

Describe the problem. (Also describe how to reproduce it, and your diagnosis and suggested correction.) Attach a listing if available.

Microsoft Use Only

Tech Support _____ Date Received _____

Routing Code _____ Date Resolved _____

Report Number _____

Action Taken:

Documentation Feedback

Please help us make our documentation better by completing and returning this card. Your feedback will help us improve future products.

Name _____

Address _____

City _____ State _____ Zip _____

Product _____ Version no. _____

Operating system _____

Computer _____ Memory (K) _____

RAM disk ☐ yes ☐ no Hard disk ☐ yes ☐ no

Programming experience: _____ years; in this language: _____years

Please rate the documentation on a scale of 1 to 5.

	5	4	3	2	1	
Complete	☐	☐	☐	☐	☐	Incomplete
Accurate	☐	☐	☐	☐	☐	Inaccurate
Organized	☐	☐	☐	☐	☐	Disorganized
Usable	☐	☐	☐	☐	☐	Unusable
Readable	☐	☐	☐	☐	☐	Unreadable
Understandable	☐	☐	☐	☐	☐	Unintelligible
Attractive	☐	☐	☐	☐	☐	Unattractive
Excellent	☐	☐	☐	☐	☐	Poor

Was installation easy to understand? ☐ yes ☐ no

Did you work through the demonstration session? ☐ yes ☐ no

If yes, was it useful? ☐ yes ☐ no

Is information easy to find? ☐ yes ☐ no

Do you use your documentation mainly for:
☐ reference? ☐ learning to program?
☐ learning to operate the software?

Did you find factual errors in the documentation? ☐ yes ☐ no
If yes, please give manual name, page number, and description of the error.

Is more information needed on how to program? ☐ yes ☐ no
If yes, on what topics?

Is the documentation intimidating? ☐ yes ☐ no

Are the manuals:
☐ too long? ☐ too short? ☐ about the right length?

Other suggestions or comments? (Use back of card if necessary.)

Comments and suggestions become the property of Microsoft Corporation.

detach here

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 108 BELLEVUE, WA U.S.A.

POSTAGE WILL BE PAID BY ADDRESSEE

MICROSOFT®
16011 NE 36th Way
Box 97017
Redmond, WA 98073-9717

Attn: Manager, Languages User Education
Bldg. 2

MICROSOFT®

04/86

tape here

MICROSOFT®